UNIVERSIDAD COMPLUTENSE DE MADRID

BACHELOR'S THESIS

Deep Learning for the classification of events from Imaging Atmospheric Cherenkov Telescopes

Author:
Jaime SEVILLA MOLINA

Supervisors:
Daniel NIETO CASTAÑO
Juan JIMÉNEZ CASTELLANOS

A thesis submitted in fulfillment of the requirements
for the degree of Bachelor of Mathematics and Computer Engineering

in the

Facultad de Informática

2018-2019 academic year

May 31, 2019

Declaration of Authorship

I, Jaime SEVILLA MOLINA, declare that this thesis titled, "Deep Learning for the classification of events from Imaging Atmospheric Cherenkov Telescopes" and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a Bachelor's degree at this University.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: 

Date: 31/05/2019

"Understanding vision and building visual systems is really understanding intelligence."

Fei-Fei Li

UNIVERSIDAD COMPLUTENSE DE MADRID

Abstract

Facultad de Informática

Bachelor of Mathematics and Computer Engineering

Deep Learning for the classification of events from Imaging Atmospheric Cherenkov Telescopes

by Jaime SEVILLA MOLINA

We develop two new computer vision models for the task of gamma-ray like event classification from Cherenkov Telescope Array (CTA) images, respectively based on LSTMs and Neural Attention.

The models are implemented and trained in our own Deep Learning framework based on the CTLearn package, and shown to outperform the State Of The Art.

Keywords: *Deep Learning, Computer Vision, LSTMs, Neural Attention, Imaging Atmospheric Cherenkov Telescopes*

* * *

Desarrollamos dos modelos de visión por ordenador para la tarea de clasificación de eventos similares a rayos gamma a partir de imágenes de la matriz de telescopios Cherenkov (CTA, de sus siglas en inglés), respectivamente basados en LSTMs y atención neural.

Los modelos son implementados y entrenados en nuestro propio entorno de aprendizaje profundo basado en la librería CTLearn. Se muestra cómo los modelos superan el estado del arte actual.

Palabras clave: *aprendizaje profundo, visión por ordenador, LSTMs, atención neural, telescopios Cherenkov de imagen atmosférica*

Acknowledgements

I would like to thank first of all my advisors. The continued advice and support from Daniel Nieto has been instrumental to the completion of this thesis. Juan Jimenez has provided me with very useful feedback to improve my work.

Secondly, the rest of the CTLearn team have been incredibly helpful and supportive, and without their previous work this whole project would not be possible. I'd like to highlight the role of Ari Brill, Bryam Kim and Tjark Miener, who have helped me a great deal through excellent discussion and their superb coding skills.

Lastly I would like to thank my family, without whom I could not be studying and who have inspired me to aim for excellence.

Contents

List of Figures

List of Tables

List of Abbreviations

AUROC	**A**rea **U**nder the curve of the **R**eceiving **O**perator **C**haracteristic
GBDT	**G**radient **B**oosted **D**ecision **T**ree
CNN	**C**onvolutional **N**eural **N**etwork
CTA	**C**herenkov **T**elescope **A**rray
DL	**D**eep **L**earning
DT	**D**ecision **T**ree
FCN	**F**ully **C**onnected **N**etwork
GPU	**G**raphical **P**rocessing **U**nit
GRU	**G**ated **R**ecurrent **U**nit
IACT	**I**maging **A**tmospheric **C**herenkov **T**elescope
LSTM	**L**ong-**S**hort **T**erm **M**emory cell
RNN	**R**ecurrent **N**eural **N**etwork
NN	**N**eural **N**etwork
LST	**L**arge **S**ize **T**elescope
MSTF	**M**edium **S**ize **T**elescope : **F**lash Camera
MSTN	**M**edium **S**ize **T**elescope : **N**ectar Camera
MSTS	**M**edium **S**ize **T**elescope : **S**CT Camera
SST1	**S**mall **S**ize **T**elescope : Single Mirror
SSTA	**S**mall **S**ize **T**elescope : **A**STRI Camera
SSTC	**S**mall **S**ize **T**elescope : **C**HEC

Dedicated to Elena Molina, for there is no wiser mother

Chapter 1

Introduction

1.1 Context

The **Cherenkov Telescope Array** (CTA) is an international observatory complex under construction with the goal of studying high-energy cosmic phenomena, particularly through the detection of gamma-rays from the ground. [Cta].

Gamma rays do not travel all the way to the ground, but instead generate upon colliding with Earth's atmosphere a Extensive Air Shower of subatomic particles that can travel faster than the speed of light in the air. Due to this fact, these cascades produce Cherenkov photonic radiation, which can be detected by the observatory's **Imaging Atmospheric Cherenkov Telescopes** (IACTs), see figure 1.2.

IACTs of several sizes and different types of cameras have been designed to cover different energy ranges. Those include the Large Size Telescopes (LSTs), Medium Size Telescopes (MSTs), equipped with FlashCams (MSTF), NectarCams (MSTN) and SCT (MSTS); and Small Size Telescopes (SSTs), equipeed with ASTRI cameras (SSTA), single-mirror cameras (SST1) and CHEC cameras (SSTC). Figure 1.3 includes information about the specifications of the different telescopes and cameras.

The telescopes will be deployed on two locations of the globe. One array will be placed in the Northern Hemisphere, in the Canary Island of La Palma, Spain, and another one in the Southern Hemisphere, close to Cerro Paranal, Chile. (see figure 1.4a). Each Extensive Air Shower is captured by several telescopes of the array, producing a stereoscopic image of the event.

Once the CTA project is complete, it will offer us an unparalleled view of the cosmos. CTA will allow us to better understand the exotic phenomena that results in the gamma rays that cross Earth's path.

FIGURE 1.1: CTA Logo [Cta]

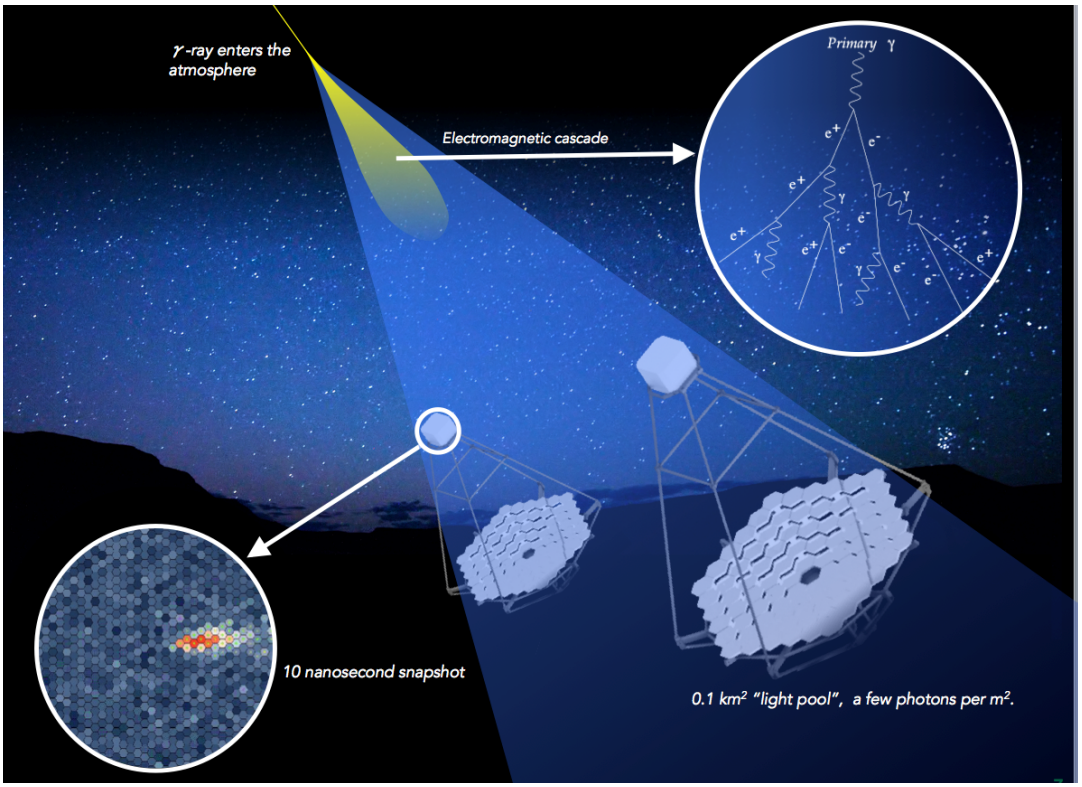


FIGURE 1.2: How CTA detects Cherenkov light [Cta]

Such events are massively energetic, and observing differences between the predictions of Relativity and those extreme cases could be fundamental to improve our understanding of fundamental physics.

In order to process the vast amount of data expected to be registered by CTA observatory we need to employ state-of-the-art machine learning techniques. The **CTLearn** project is currently working on the development of such techniques.

Their current focus is on the task of IACT event classification - that is, identifying when a photonic event received by the CTA has been caused by a gamma ray.

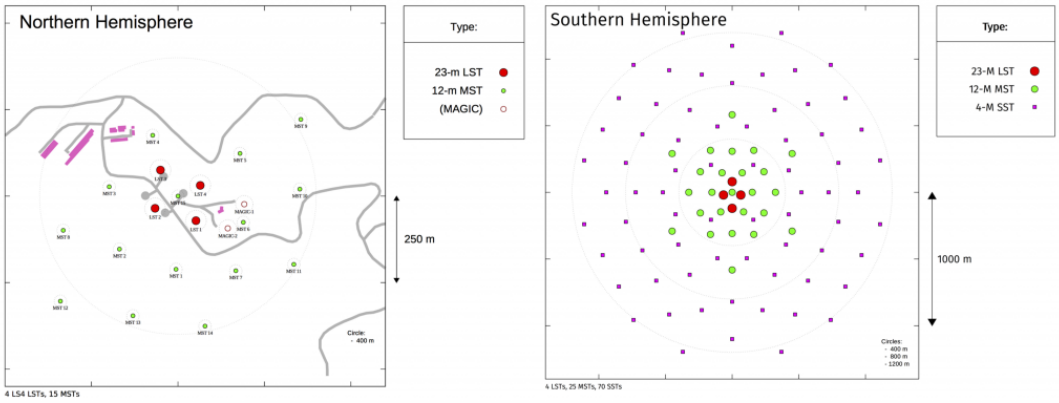
CTA registers both hadronic and gamma-ray induced showers. The enormous quantity of data cannot be vetted in real time, so it is saved to disk and later processed my Machine Learning models that perform background suppression to isolate the relevant events as a prerequisite for further analysis, such as energy level and arrival direction regression.

This task is specially relevant when taken into account that gamma ray cascades are extremely rare, happening at a estimated rate of between 1 to 100 photons originated from a gamma ray event per century and square meter of the Earth [Cta] while hadronic events are far more common - highly sensitive and well calibrated models are needed to not miss crucial data.

Our work will focus on IACT event classification as well. We will seek to build upon and improve CTLearn's current models.

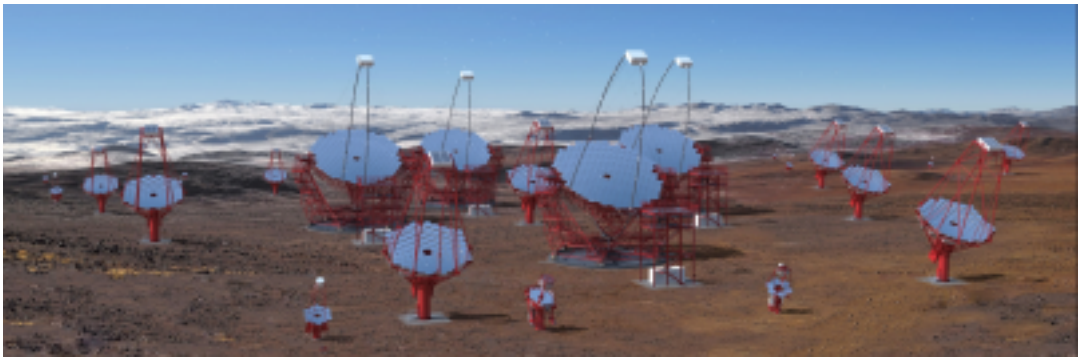
	Large-Sized Telescope (LST)	Medium-Sized Telescope (MST)			Small-Sized Telescope (SST)		
		FlashCam	NectarCam	SCT	ASTRI	GCT	SST-1M
Required energy range	20 GeV – 3 TeV	80 GeV – 50 TeV			1 TeV – 300 TeV		
Energy range (in which subsystem provides full system sensitivity)	20 GeV – 150 GeV	150 GeV – 5 TeV			5 TeV – 300 TeV		
Number of telescopes	4 (South) 4 (North)	25 (South) 15 (North)			70 (South) 0 (North)		
Optical design	Parabolic	Modified Davies-Cotton		Schwarzschild-Couder	Schwarzschild-Couder		Davies-Cotton
Primary reflector diameter	23.0 m	11.5 m		9.7 m	4.3 m	4.0 m	4.0 m
Secondary reflector diameter	--	--		5.4 m	1.8 m	2.0 m	--
Effective mirror area (including shadowing)	370 m^2	88 m^2		41 m^2	8 m^2	8.9 m^2	7.5 m^2
Focal length	28 m	16 m		5.6 m	2.15 m	2.28 m	5.6 m
Total weight	103 t	82 t		80 t	19 t	11 t	8.6 t
Field of view	4.3 deg	7.5 deg	7.7 deg	7.6 deg	10.5 deg	8.3 deg	8.8 deg
Number of pixels in Cherenkov camera	1855	1764	1855	11328	2368	2048	1296
Pixel size (imaging)	0.1 deg	0.17 deg	0.17 deg	0.067 deg	0.19 deg	0.17 deg	0.24 deg
Photodetector type	PMT	PMT	PMT	SiPM	SiPM	SiPM	SiPM
Telescope readout event rate (before array trigger for MSTs and SSTs)	>7.0 kHz (after LST array trigger)	>6 kHz	>7.0 kHz	>3.5 kHz	>0.3 kHz	>0.4 kHz	0.6 kHz
Telescope data rates (readout of all pixels; before array trigger)	24 Gb/s	12 Gb/s			2 Gb/s		3.2 Gb/s
Positioning time to any point in the sky (>30° elevation)	30 s	90 s			60 s		
Pointing precision	<14 arcseconds	<7 arcseconds		<10 arcseconds	<7 arcseconds		
Observable sky	Any astrophysical object with elevation > 24 degrees						

FIGURE 1.3: Information about the different CTA telescopes [Cta]



(A) Planned locations of the Telescope Arrays

(B) Northern Hemisphere Site rendering



(C) Southern Hemisphere Site rendering

FIGURE 1.4: Plans for the CTA construction [Cta]

1.2 Previous work

The task of automated IACT event classification has been attempted in the past using different approaches.

In this section we briefly survey some of the most important ones and give a quick introduction to Deep Learning, the family of machine learning methods upon which CTLearn is built.

1.2.1 Decision Trees

Decision Trees (DTs) are a supervised machine learning classification method where a tree is constructed which gives guidelines to classify an example from its features, typically extracted from a paramerization of the shower image.

To construct the tree, we recursively divide the training data on two groups ("leaves") by splitting by a threshold in some feature such that the combined entropy of the resulting leaves is minimized. Ideally the choice of the feature to split on is also made according to the entropy minimization criteria, but quite often we will choose a random feature to decrease computational costs.

To use the resulting tree for classification of an event, we start at the root and follow the splits according to the event features, and upon arriving to a leaf we assign to the event the class of the majority of training examples which compose the leaf.

In the context of IACT event classification, the features used for classification are manually chosen statistics of the image data, such as size, width, length, total energy received and image noise.

For example, one split may query us on the estimated width of the event, telling us to go down one branch or the other depending on whether the width exceeds a certain threshold. This process of querying and taking branches is repeated until we arrive at a leave node, where there is a class label indicating the prediction of the tree for this particular event. See an example of such a tree in 1.5.

Typically, single decision trees are overfitted to the training data and produce inaccurate classifications. To address this issue we can resort to a refinement of the DT method called **Random Forests (RFs)**.

To build a RF classifier we create a collection of DTs, each of which is trained on a different random subset of the total training data.

Let us denote as $T_i(x)$ the label (0 or 1) assigned to example x by tree i out of n of the RF. The classification procedure then assigns a probability to the event x pertaining to the positive class equal to $RF(x) = \frac{\sum_i T_i(x)}{n}$.

The RF approach for IACT event classification was explored in *Implementation of the Random Forest Method for the Imaging Atmospheric Cherenkov Telescope MAGIC* [Alb+08]. Random Forest is the current background suppression method used by the MAGIC operators.

Another refinement of the DT method are **Gradient Boosted Decision Trees (GBDTs)**.

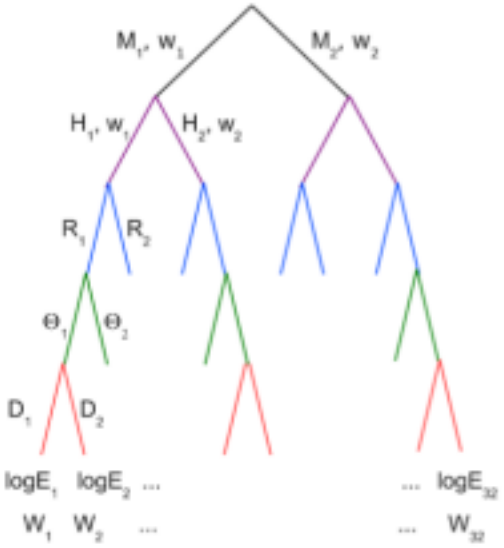


FIGURE 1.5: Schematic of the Decision Tree used for IACT event classification in [Bec+11]

GBDTs are ensemble models built sequentially. We first build a DT of a pre specified depth fitting some training data. Then the next tree is trained to predict the discrepancy between the first tree and the ground truth, so that the sum of both trees produces a more accurate prediction.

The following trees will be built on the same principle, trained on the discrepancy between the sum of the previous trees and the ground truth.

Our final model will end up making a classification equal to the sum of the predictions of each tree we have built.

$$GBDT(x) = T_1(x) + T_2(x) + ... + T_n(x)$$

For further reference, a visual, interactive explanation of GBDTs can be found in [Rog16].

Several papers have been written about the applications of GBDTs to IACT event classification, which differ on their choices of classification features [KPM17] [OvE09] [Bec+11].

1.2.2 Deep Learning

When we talk about Deep Learning we are referring to a family of machine learning techniques based on a particular type of function approximators, called **Neural Networks (NNs)**, built using several layers of computing units, called cells.

Cells in each layer of a Neural Network compute a function applying a linear combination of the results from the previous layer concatenated with a derivable nonlinearity that allows the whole network to learn how to imitate complicated functions.

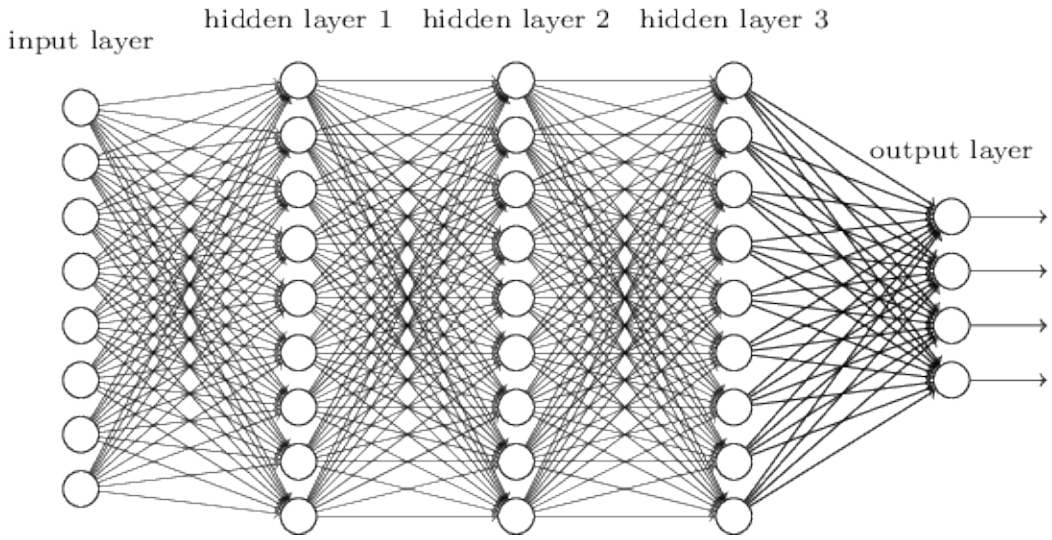


FIGURE 1.6: Deep Neural Network schematic [Nie18]

Neural Networks are initialized with random parameters on the linear weights used by each cell, but using a training set of labelled examples we can train the NN to approximate the logic that associates each example with its corresponding label.

For this purpose, we feed the result of the neural network through a derivable loss function, which represents in a single number (usually non negative) how correct the results produced by the neural network are on each example of the training set.

Since all components of the neural network and the loss function are derivable, we can compute the gradient of the loss function with respect to the NN parameters, which can be efficiently computed using the **backpropagation** technique. Then we can use **gradient descent** to update the parameters toward values that better approximate the relation between examples and labels.

Careful training will result in a NN which will correctly generalize the example-label relation in the training set to examples it has not seen before, resulting in powerful and adaptable systems with many applications.

Deep Learning techniques have allowed us to tackle artificial intelligence problems of great importance and variety, and for which classical machine learning methods struggled to deal with.

Even though the ideas behind Deep Learning are relatively old [IL73], the recent advances in computer design have removed the key bottleneck to unlock great performance in NN training: computational power.

Some landmark applications of neural networks:

- In the domain of **computer vision** deep Convolutional Neural Networks have been shown to outperform all previous methods for classifying images from the ImageNet benchmark [KSH12]
- In the domain of **natural language processing** Recurrent Neural Networks are the State of the Art for translation [SVL14], text prediction [Rad+19] and other NLP tasks.

- In **game strategy** we have recently seen many strong results using Deep Reinforcement Learning architectures trained via self-play in games as diverse as Go [Sil+16] and Starcraft [Vin+19] .

Different NN architectures have been developed to be applied to different types of data and tasks. Perhaps the three most important families of architectures, which are also the ones that are most relevant for our project, are Fully Connected Neural Networks, Convolutional Neural Networks and Recurrent Neural Networks.

Fully Connected Networks (FCNs) are the bread-and-butter of Deep Learning. They are made out of neurons which compute the result of applying a non linearity to a linear combination of all the activations of neurons from the previous layer. Figure 1.6 represents a NN made out of four Fully Connected Layers.

Convolutional Neural Networks (CNNs) have been used and continue to be used to great success as image feature extractors. Their implementation is based on **discrete convolutions**, where a filter of weights is successively overlaid upon all possible positions in the image and multiplied to get a pixel output. They can be thought of as FCNs where parameters are reused to analyze each patch of the image.

Recurrent Neural Networks (RNNs) have been designed to process input data of variable length, such as sentences or, in the case that concerns us, collections of images of the same IACT event. RNNs work by creating a NN of variable length whose parameters are periodically repeated, then used this "unrolled" network for training or prediction on a particular batch of examples. The two most common RNNs architectures are **Gated Recurrent Units** (GRUs) [Cho+14] and **Long-Short Term Memory** (LSTMs) cells [HS97].

Those three building blocks can be used together to build complex architectures capable of learning sophisticated tasks.

LSTM walkthrough

In this section we describe the mathematical details behind a typical CNN-RNN architecture such as the one we will be implementing in our project.

The CNN extracts the features from the images. It is composed of **convolutional** and **maxpool** layers.

2D convolutional layers apply discrete convolutions over a 2D image of activations with several channels.

They are parameterized by a series of **filters**. Each filter is a volume of weights of depth equal to the number of channels of the input, that is convolved with the input to obtain one channel of the next activation.

The channels are padded with zeroes to preserve their size after each convolution.

After the convolution, the image is fed through a pointwise non-linear function.

The kind of activation used in each layer is customizable, but through this paper we will stick to the rectifier function, defined as follows:

$$f(x) = \left\{ \begin{array}{ll} x, & \text{for } 0 \leq x \\ 0, & \text{for } x < 0 \end{array} \right\} \tag{1.1}$$

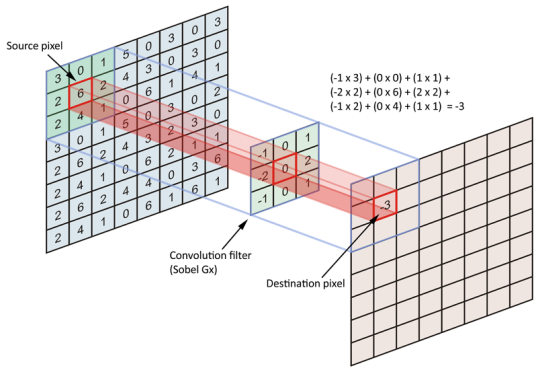


FIGURE 1.7: Visualization of the convolution operation (unknown author)

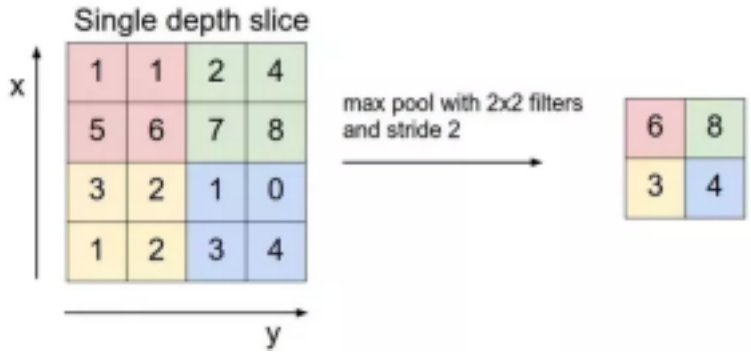


FIGURE 1.8: Max pooling downsampling example [Cs2]

To reduce the size of the activation volumes, we use **maxpool** layers. They lack trainable parameters, and are characterized by a hyperparameter k indicating the downsampling factor.

To apply a maxpool, each channel of the input activation is divided into equal regions of size $k \times k$, and each region is mapped to a single number equal to the max value of the activation (see figure 1.8).

After the CNN, we unroll all the resulting activations of the final volume on top of each other to create a vector of activations suitable to be fed to the next part of the NN (see figure 1.9).

The LSTM layer combines the features from different images.

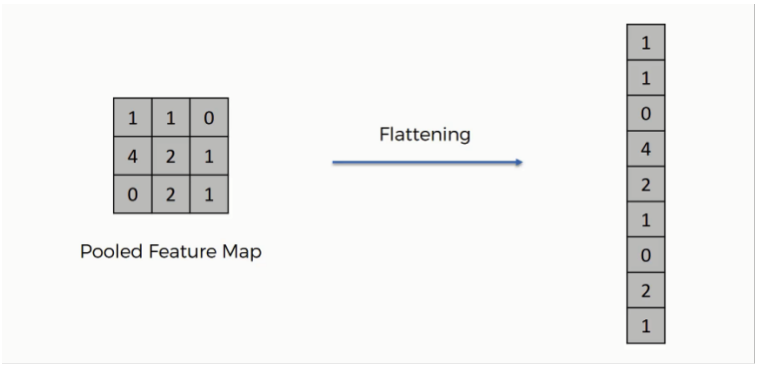


FIGURE 1.9: Flatten operation example (unknown author)

It is composed of a series of cells, as many as the number of images to be processed.

Each cell takes as input the feature vector from the CNN, and the memory vector and output of the previous cell (in the first cell, the memory and previous output inputs are set to zero). It produces both an output and an updated memory vector.

The internal logic of each LSTM cell is controlled by a series of gates and generators that dictate how to update the memory and produce the output. Each valve / generator is a one-layer neural network on its own.

First of all we have the **forget gate** and the **update gate**, which take as input the feature vector, the previous memory vector and the previous output. Their output is a single number between 0 and 1. The forget gate f_t determines how much of the previous memory block will be retained, while the update gate u_t determines how much to update the old memory.

The new memory proposal M_t is generated by a neural network that takes as input the previous output and the current feature vector.

In equation form:

$$C_t = C_{t-1} \cdot f_t + M_t \cdot u_t \tag{1.2}$$

Where C_t is the memory block of the tth cell, f_t is the forget gate, M_t is the proposal for the memory update and u_t is the update gate.

To generate the output, an output proposal O_t is generated by a NN which processes both the previous output and the current feature vector. The result is multiplied by the content of the updated memory block through a non-linearity to produce the cell output h_t.

$$h_t = O_t \cdot f(C_t) \tag{1.3}$$

Figure 1.10 summarizes the behaviour of each LSTM cell.

The FCN processes the output of the final cell of the LSTM layer and outputs the final prediction. It is composed of several fully connected layers followed by a sigmoid activation.

Each fully connected layer has associated an activation function f, a weight matrix W and a bias vector b that govern the transformation from one layer of activations to the next one, following this equation:

$$a^{[n]} = f(W^{[n]} a^{[n-1]} + b^{[n]}) \tag{1.4}$$

Again, the type of non-linearity f can be customized, but we will be using rectifiers.

The last activation is always a **softmax function** σ. The output of the sigmoid is always a vector whose components are between 0 and 1 and sum to 1. We will interpret this output as a probability vector $\hat{y}$, each of whose components represents the probability that the event processed belongs to the index class.

$$\hat{y} = \sigma(W^{[n]} a^{[n-1]} + b^{[n]}) = \frac{1}{1 + e^{-W^{[n]} a^{[n-1]} - b^{[n]}}} \tag{1.5}$$

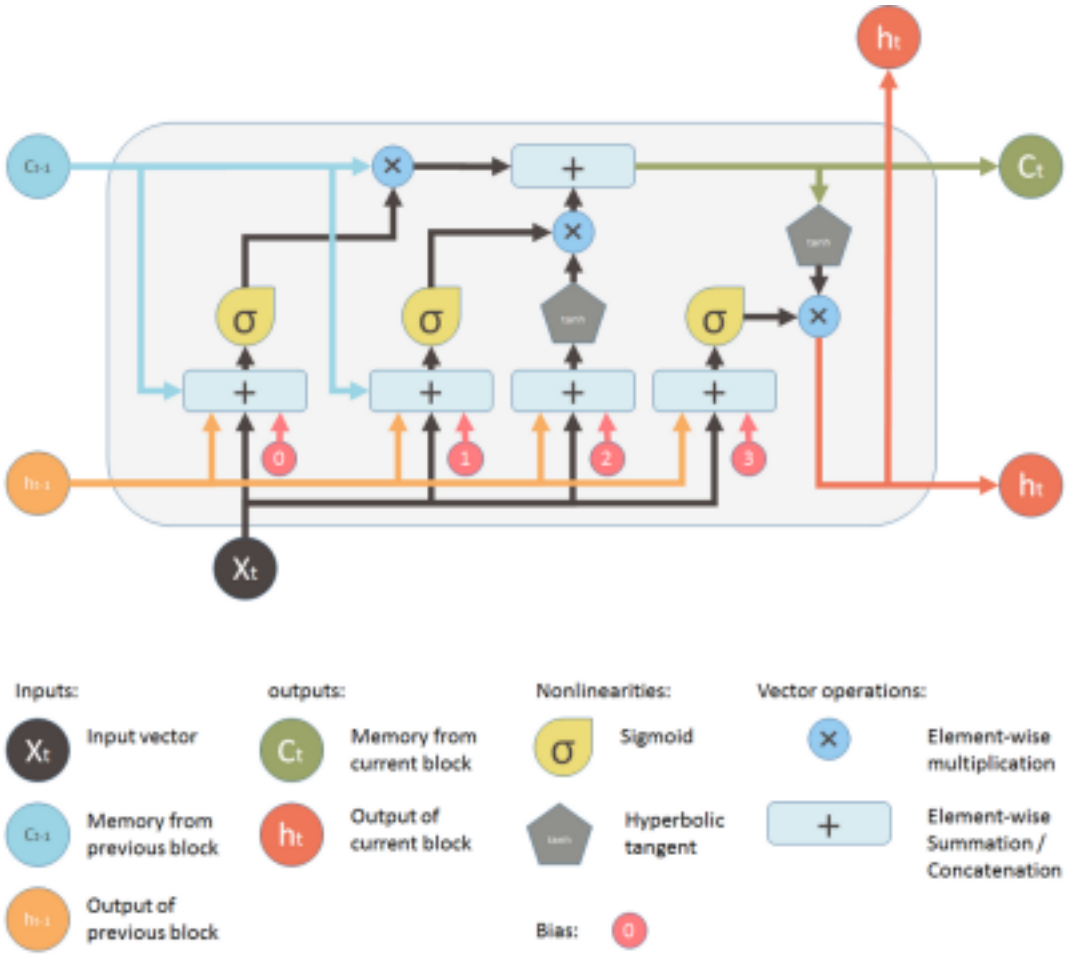


FIGURE 1.10: LSTM cell diagram
LSTM cell diagram [Yan16]

Training a Deep Learning architecture

The initial architecture can be thought of as a function which depends diferentially on a set of parameters θ, which include the weights and bias terms of each layer.

Those parameters are initialized randomly, and subsequently the predictions made by the resulting NN are wildly off.

In this section we describe the procedure by which we will find a choice of parameters that will properly capture the structure of the classification problem at hand.

Loss function In order to compare the performance of different choices of parameters, we need to evaluate the performance of the architecture over our training set of examples. Furthermore, we need this evaluation to be a single real number so we can quickly compare two choices of parameters and determine which one is best.

This mapping from the training set and a choice of parameters to a single number is called a **loss function**.

In our case, the loss function we will be using will be the **cross-entropy loss**, defined as follows.

$$L(y, \hat{y}(X, \theta)) = -y \log(\hat{y}(X, \theta)) - (1 - y) \log(1 - \hat{y}(X, \theta)) \tag{1.6}$$

In this equation, y is the ground truth of the training set, while $\hat{y}$ are the predictions made by our NN over the training set examples and depends differentially on the choice of parameters θ and the set of features X.

Backpropagation The procedure for improving the choice of parameters over the training set is straightforward: since the loss function L depends differentially on the choice of parameters, we can take the derivative over θ and update the parameters on the opposite direction to $\frac{\partial L}{\partial \theta}$. This is called **gradient descent**.

Computing the derivatives for each parameter is expensive, but we can take advantage of the chain rule of calculus to do as few operations as possible, in a process called **backpropagation**.

$$\frac{df}{d\theta} = \frac{\partial f}{\partial g} \frac{\partial g}{\partial \theta} \tag{1.7}$$

This is better illustrated by an example.

Let us suppose a three layer FCN, described by these equations:

$$z^{[1]} = W^{[1]}a^{[0]} + b^{[1]} \tag{1.8}$$
$$a^{[1]} = f(z^{[1]}) \tag{1.9}$$
$$z^{[2]} = W^{[2]}a^{[1]} + b^{[2]} \tag{1.10}$$
$$a^{[2]} = f(z^{[2]}) \tag{1.11}$$
$$z^{[3]} = W^{[3]}a^{[2]} + b^{[3]} \tag{1.12}$$
$$\hat{y} = a^{[3]} = \sigma(z^{[3]}) \tag{1.13}$$
$$\tag{1.14}$$

We want to compute the derivatives of the loss function L with respect to the weight and bias terms:

$$\frac{\partial L}{\partial W^{[3]}} = \frac{\partial L}{\partial a^{[3]}}\frac{\partial a^{[3]}}{\partial W^{[3]}} = L'(a^{[3]})\sigma'(z^{[3]})a^{[2]} \tag{1.15}$$
$$\frac{\partial L}{\partial W^{[2]}} = (\frac{\partial L}{\partial a^{[3]}})\frac{\partial a^{[3]}}{\partial a^{[2]}}\frac{\partial a^{[2]}}{\partial W^{[2]}} = L'(a^{[3]})\sigma'(z^{[3]})W^{[3]}f'(z^{[3]})a^{[1]} \tag{1.16}$$
$$\frac{\partial L}{\partial W^{[3]}} = (\frac{\partial L}{\partial a^{[3]}}\frac{\partial a^{[3]}}{\partial a^{[2]}})\frac{\partial a^{[2]}}{\partial a^{[1]}}\frac{\partial a^{[1]}}{\partial W^{[3]}} \tag{1.17}$$

In the equations, the terms in parenthesis have been calculated in a previous equation, and thus can be reused.

This reutilization of terms is the core of the backpropagation algorithm.

After the gradients are computed we can update the weights of our neural network following the next equation.

$$\theta_{t+1} := \theta_t - \eta\nabla_\theta L$$

Where θ are the parameters of the net, L is the loss function and η is the learning rate. To improve the training quality, the learning rate does not stay constant through the training process, but instead it updates according to the optimizer we choose.

During this project we will resort to the Adam (Adaptive moment estimation) optimizer [KB14], where the learning rate evolves according to the following rule:

$$m_t := \beta_1 m_{t-1} + (1-\beta_1)\nabla_\theta L$$
$$v_t := \beta_2 v_{t-1} + (1-\beta_2)(\nabla_\theta L)^2$$
$$\hat{m}_t := \frac{m_t}{1-\beta_1^t}$$
$$\hat{v}_t := \frac{v_t}{1-\beta_2^t}$$
$$\eta_t = \frac{\eta}{\sqrt{\hat{v}_t}}\hat{m}_t$$

m_t and v_t are the estimates of the first and second moment (mean and uncentered variance), while $\hat{m}_t$ and $\hat{v}_t$ are their respective bias-corrected versions.

We do not need to implement backpropagation nor the Adam optimization algorithm ourselves; the Tensorflow backend will take care of it for us.

Regularization To prevent our model from overfitting to the training data, we resort to what are know as regularization mechanisms. We now present three commonly used regularization mechanism that we will use in our project.

The first one is using **L2 kernel regularizers** in our layers.

These are additional terms of the loss function which penalize using extreme weights, as measured by the euclidean magnitude of the weights seen as a n-dimensional vector.

The L2 regularized loss equation is as follows:

$$L(y, \hat{y}(X, \theta)) = -y \log(\hat{y}(X, \theta)) - (1 - y) \log(1 - \hat{y}(X, \theta)) + \lambda |\theta|^2 \tag{1.18}$$

where λ is a constant positive hyperparameter.

The second regularization mechanism we use is to employ **dropout** layers between each layer of our model.

They are inactive when the NN is used for predictions, but during training they randomly 'drop' some activations to zero.

This forces the NN to learn to not rely on specific features, and the result is a more robust model with a better performance on examples never seen before [Sri+14].

The third regularization mechanism is **early stopping**.

When a model has been subject to backpropagation for long enough with the same training set, it starts **overfitting**. That is, it learns the particular idiosyncrasies of the training set and memorizes their corresponding label instead of learning the general principles that will allow the model to perform well even in cases it has not seen before.

In order to prevent this, we monitor the accuracy of our models in the validation set after each epoch of training, and stop the training whenever we fail to detect and improvement over the previous epoch.

Iterated Grid Search The previously described training methods will allow us to find an appropriate value for the differentiable weights of the model, but we still need to choose some hyperparameters that are not differentiable.

For this we will use iterated grid search.

Grid search is a basic hyperparameter procedure where we try every possible combination of parameters over a discrete range of values, run a training session for each and finally stick to the one which achieves best performance over the validation split.

Due to a combinatorial explosion, we cannot tune all the hyperparameters at the same time, so we will work iteratively. The first stage will consist of a coarse grid

FIGURE 1.11: CTLearn Logo [AB18]

search to find a baseline model, and then we will try changing the configuration values of this baseline model one by one seeking improvement.

Deep Learning applied to IACT

The idea of applying Deep Learning techniques to IACT event classification was first considered by Nieto et al. in *Exploring deep learning as an event classification method for the Cherenkov Telescope Array* [Nie+17].

Later in *Application of Deep Learning methods to analysis of Imaging Atmospheric Cherenkov Telescopes data* [Shi+18] a CNN + RNN model was employed, and it was found to outperform a state-of-the-art GBDT method.

Unlike with DT-based approaches, DL methods in IACT event classification can use (after an appropriate conversion from the hexagonal grid images to square pixel data) the raw information per pixel captured by the telescopes instead of relying on hand crafted features, which explains their superior performance.

1.2.3 CTLearn

CTLearn is a Python package under development that uses the deep learning technique to analyze data from IACT arrays.

CTLearn includes modules for loading and manipulating IACT data and for running machine learning models with TensorFlow, using pixel-wise camera data as input. Its high-level interface provides a configuration-file-based workflow to drive reproducible training and prediction.

We will be working starting from the v0.2.0 release of CTLearn, implemented using Python 3.6.5 and Tensorflow 1.9.0.

Its full dependencies are listed below:

```
name: ctlearn
channels:
    - conda-forge
    - menpo
dependencies:
    - python=3.6.5
```

Telescope type	Val images	Positive	Negative	AUROC	Accuracy	Training time
LST	17960	8976	8984	0.7887	70.38%	41m 22s
MSTF	74032	35753	38279	0.8360	74.60%	55m 0s
MSTN	85821	41524	44297	0.8659	78.04%	58m 10s
MSTS	60222	30602	29620	0.8709	78.57%	59m 4s
SST1	42180	21365	20815	0.8542	77.11%	45m 30s
SSTA	44986	22757	22229	0.8105	72.59%	42m 17s
SSTC	43626	21757	21869	0.8118	73.90%	42m 4s

TABLE 1.1: Validation results for CTLearn single-telescope CNN model, v0.2.0 benchmark

```
- matplotlib
- numpy
- opencv3
- pillow
- pip
# TensorFlow-GPU v1.9.0:
- pip:
    - https://storage.googleapis.com/tensorflow/linux/gpu/
      tensorflow_gpu-1.9.0-cp36-cp36m-linux_x86_64.whl
- pytables
- pyyaml
- scikit-learn
- scipy
```

Release 0.2.0 of CTLearn includes three Deep Learning models suited for the task of event classification using data from Imaging Atmospheric Cherenkov Telescopes as input:

- The **Single Tel** convolutional model is capable of classifying single-telescope images. It passes images through a CNN which then feeds into a FCN.
- The **Variable Input Network** model takes as input arrays of n (for a value of n chosen beforehand) images of the same event from different telescopes (of the same type). Those images are fed into respective CNN blocks (with weights shared) and the combined output is fed into a FCN (see figure 1.12).
- The **CNN-RNN** model takes the same input as the Variable Input Network. Its architecture consists of a fixed number of CNN blocks with shared weights which then feed into respective LSTM units. The combined output of the LSTM units goes through a FCN which produces the prediction (see figure 1.13).

Current classifications results are benchmarked (see tables 1.1 and 1.2) with training and validation on a medium size (179G GB, 377098 events, 3688224 images in total) set of Montecarlo IACT simulated data [Ber+13]. We will use this benchmark as a baseline for our research, and use the same dataset for training our models.

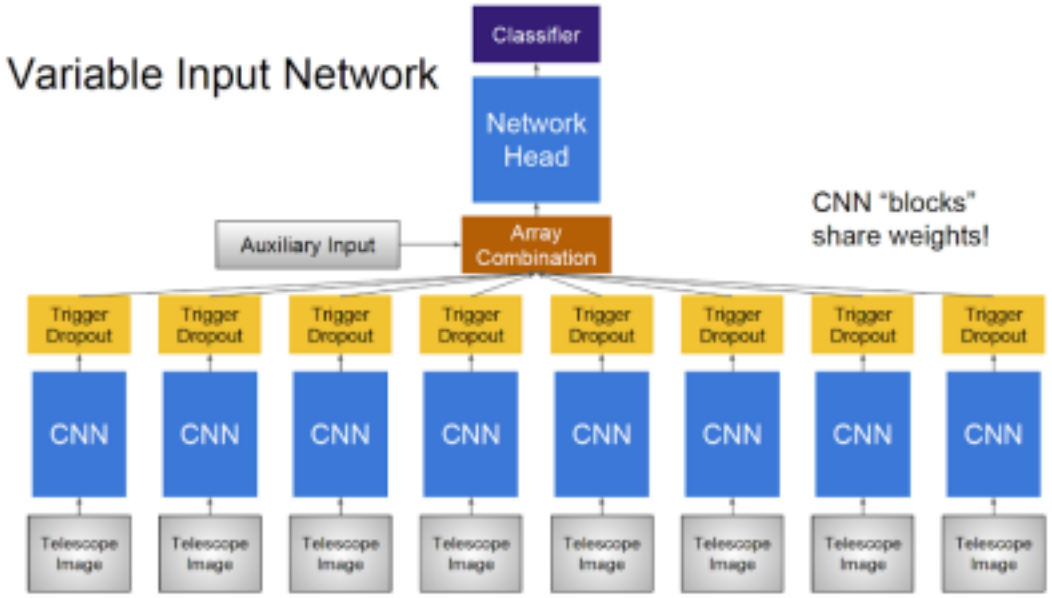


FIGURE 1.12: Variable Input Network CTLearn model [Bri18]. A fixed number of image features extracted with a CNN are stacked to produce a embedding of the image sequence which is used for classification

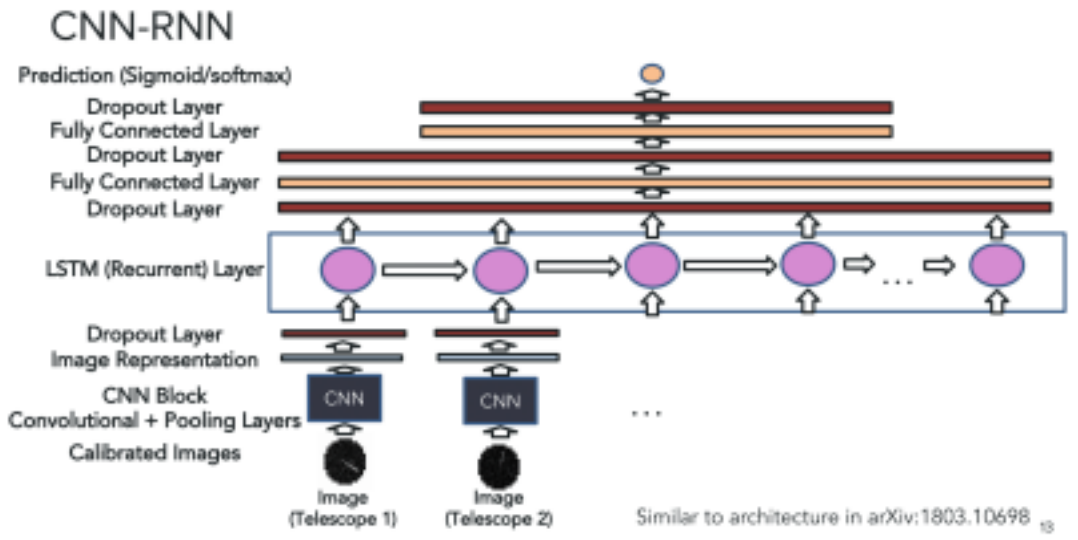


FIGURE 1.13: CNN-RNN CTLearn model [Bri18]. In this case, the image features are fed through a statically unrolled LSTM layer to produce the output.

Telescope type	Events	Positive	Negative	AUROC	Accuracy	Training time
LST	8541	3950	4591	0.8285	73.43%	0:37:14
MSTF	24982	12154	12828	0.8961	80.23%	2:06:32
MSTN	26937	12861	14076	0.9169	83.10%	2:15:52
MSTS	22306	11123	11183	0.9048	81.18%	2:37:34
SST1	19788	9423	10365	0.8997	81.47%	3:21:48
SSTA	18367	9399	8968	0.8556	75.27%	2:10:55
SSTC	19064	9787	9277	0.9072	80.64%	1:51:05

TABLE 1.2: Validation results for CTLearn CNN-RNN model, v0.2.0 benchmark

1.3 Goals

Our goal will consist on implementing a new model for gamma-ray like event classification using arbitrarily many images as an input from different types of telescopes, thus overcoming a key limitation of the current CNN-RNN model.

Currently, the CNN-RNN model is restricted to processing a preset number of images at a time, all from the same type of telescope. Each image is fed through a CNN extractor, and the resulting features are fed through a LSTM cell unrolled n times. The results of this process are concatenated and fed through a FCN, which makes the final class prediction (see figure 1.13).

The hope is that enabling the use of extra images will allow us to improve the event classification score in the array case, while still robustly identifying single images.

The formal task will consist in assigning to each batch of images from the same event but different telescopes a number between 0 and 1, which we will interpret as representing the probability of the event being a positive example (that is, the event had been caused by a gamma ray) versus it being a negative example (that is, the event had been caused by a hadron).

To achieve this we will experiment with two different architectures: a LSTM model and an attention model.

In standard implementations of RNNs, the LSTM cell is unrolled as many times as images are there to be processed, and only the output of the last cell is fed to the FCN. A model built this way can applied to batches of images of arbitrary size, from single images to big collections, without needing additional training.

You can see a schematic representation of the LSTM model we intend to implement in chapter 3, figure 3.1.

Our second idea to enable classification from arbitrarily many images is using an attention mechanism. Attention is a key development in Deep Learning that enables neural networks to learn what features they should pay more attention to. In [Vas+17] Vaswani et al. suggest substituting traditional RNNs by attention mechanisms for sequence processing, and quite recently Open AI has greatly surpassed the state of the art in language processing using attention-based mechanisms [Rad+18].

We intend to implement a simple attention mechanism that will allow the combination of the features from multiple images. See the schematics for this model in chapter 4, figure 4.1.

In order to implement these models, we will work on a Deep Learning framework based off CTLearn that modernizes it code base using Keras, the open source, high-level API for Deep Learning [Cho15]. An implementation of Keras is included in TensorFlow [Aba+15].

1.4 Methodology

In this section we describe in detail the implementation, training and cross-validation plan.

1.4.1 Implementation of the new CTLearn-Keras framework

The new code will be released in the repository github.com/Jsevillamol/ctlearn. The code will be open sourced under a BSD 3-Clause "New" or "Revised" License.

Models implemented in CTLearn support a train mode and a predict mode. In train mode training steps are alternated with validations over a validation step to record progress. Predict mode is meant for users to get the predictions on new data from an already trained model.

In the v0.2.0 release, the package supports processing data from the following types of telescopes: LST, MSTF, MSTN, MSTS, SST1, SSTA, SSTC.

Runs are fully configurable via yaml files. See `config/example_config.yaml` for a documented example explaining the configuration options.

A salient option is the possibility of employing hyperparameter grid search via the `run_multiple_models.py` script.

We will focus our efforts in implementing the training and model building functionality of the project, but using Keras as the core.

The underlying mechanism to load training data will also undergo a significant overhaul, thought the overall data pipeline will be unchanged.

A detailed UML diagram describing the structure of the new framework can be found in chapter 2, figure 2.1.

1.4.2 Implementation of the models

We will provide two customizable implementations of the models we will be working with (LSTM and Attention) in yaml configuration files, convertible to h5 Keras models using functionality from our repository.

1.4.3 Training

Training and testing will be performed on the UCM GAE server on a RTX 2080 Ti GPU using the data from the v0.2.0 CTLearn benchmark.

The benchmark data is stored on .h5 files. Each file contains the results of a series of simulations of events of the same class (gamma ray events or hadron events). To each event correspond a series of images, one per each simulated telescope that detected the event.

Concretely, each file specifies:

- Class of events simulated
- Physical arrangement of the simulated telescope array, including the type and position of each telescope.
- A list of the events simulated, including labels for the type of event simulated, the energy of the event and the arrival direction of the event (altitude and azimuth).

Total events	377098
Gamma events	189633
Proton events	187465

TABLE 1.3: Statistics of the v0.2.0 benchmark dataset

Telescope type	Number	Images	Positive	Negative
LST	4	179591	89165	90426
MSTF	25	740320	360787	379533
MSTN	25	858206	414502	443704
MSTS	25	602212	307498	294714
SST1	70	421791	213795	207996
SSTA	70	449852	221810	228042
SSTC	70	436252	217940	218312

TABLE 1.4: Contents of the v0.2.0 benchmark dataset

- One table per telescope type containing the images of each event received by each telescope of that type. The images contain a channel with the integrated charge pulse per pixel received during the event and a second channel containing the peak energy arrival time per pixel since the beginning of the event.

The benchmark dataset contents are summarized in tables 1.3 and 1.4.

The training : validation split that will be used is 90 : 10, with a fixed random seed to always produce the same split.

The model weights will be trained using backpropagation. Grid search will be used for hyperparameter tuning.

To mitigate the combinatorial explosion that happens when trying to optimize over many hyperparameters at once, we will resort to an iterative grid search. First phase of the grid search will focus on achieving the highest possible performance on the training set, while the second phase of the grid search will tune the regularization parameters to mitigate the effect of overfitting.

1.4.4 Cross-validation

Testing will be performed computing performance metrics over a validation split of the v0.2.0 benchmark of CTLearn. We recall that the validation split consists of 10% of the benchmark data.

Due to discrepancies in the implementation, the validation dataset will not be exactly the same in the new framework, but we expect those results to be comparable nonetheless.

The main metrics of performance we will be using to compare models are accuracy and the Area Under the curve of the Receiving Operator Characteristic curve (AUROC).

Accuracy is defined as the ratio of events correctly classified over the total number of events.

The Receiving Operator Characteristic (ROC) curve is created by plotting for every classification threshold between 0 and 1 the proportion of examples classified as positives which were actually positive (True Positive Rate) against the proportion of examples classified as positive but were actually negative examples (False Positive Rate).

A perfect classifier would have a ROC curve which is constantly one, and thus it would delineate an area of surface one. In general, the higher the area under the ROC, the better the classifier.

Our baseline for comparison will be the results of the single-telescope and CNN-RNN models of CTLearn v0.2.0 when trained and evaluated on the validation set of the v0.2.0 CTLearn benchmark, respectively in tables 1.1 and 1.2.

Our goal is to compare our models on these same metrics and dataset to evaluate their relative performance.

Chapter 2

Framework

2.1 Phases of design

As part of the project we resolved to refactor the CTLearn framework using a new tech stack based on Keras.

This is meant to serve as a proof of concept of an alternate design that may prove easier to maintain and work with in the future of the CTALearn project.

In this chapter we focus on our journey towards the implementation. We opted by using an agile work cycle, and what is seen in this chapter is a summary of the final design decisions reached. The code can be found in github.com/Jsevillamol/ctalearn, under release v0.3.0.

2.2 Requisites

We identified the following functional requisites for the new framework:

1. Building and training a model as specified in a .yaml file
2. Building an untrained model using the model configuration in a .yaml file
3. Training pre built and perhaps pre trained models using the training configuration specified in a .yaml file
4. Producing appropriate summaries and graphs recording the improvements during the training sessions
5. Performing hyperparameter grid search

To facilitate reproducibility, all functionality should be programmable via a single .yaml configuration file.

A detailed explanation of all the functions supported by the final version program can be found in the yaml example configuration file distributed with the framework. We reproduce the contents of this file here:

```
# The data config section configures the data loader
data_config:
    # file_list_fn: path to a txt file where each
    #               line is the absolute path to a
    #               .h5 file containing data
    #               from an event
    file_list_fn: data.txt
```

```
# channels: list of image-like features per image
#           that will be loaded. Supports:
#              image_charge: total charge received by
#                            pixel during the event
#              peak_times: time in seconds when the
#                          peak of energy occured
# CAUTION: ctalearn assumes that image_charge is
#          always the first channel
channels: [image_charge]

# img_size: [img_width, img_length]
img_size: [108, 108]

# selected_tel_types: list of telescopes to load
# supported types: 'LST', 'MSTF'
selected_tel_types: ['LST']

# data_type: select between loading single
#            images or sequences
# supported modes: 'single_tel', 'array'
data_type: 'array'

# min_triggers_per_event: filter events with
#                         few telescope triggers
min_triggers_per_event: 1

# image_mapping_config: configuration for the
#                       mapping of images to arrays
image_mapping_config:
    # hex_conversion_algorithm:
    #     supported options: oversampling
    hex_conversion_algorithm: oversampling

# preprocessing_config: configuration for the
#                       preprocessing of images
preprocessing_config:
    # normalization
    # Supported modes:
    #    null: no normalization
    #    log_normalization
    normalization: null

    # resize_mode
    # supported modes: 'interpolate'
    resize_mode: 'interpolate'

    # event_order_type: order of events in array mode
    # Supported modes
    #     null: no reordering
    #     size: order images by sum of charges
```

```
        event_order_type: size

        # event_order_reverse:
        # if true, the images inside an event will be
        # ordered greatest to lowest
        event_order_reverse: true

        # min_imgs_per_seq: pads sequences smaller than
        #                   this with zero images
        min_imgs_per_seq: 4
        # max_imgs_per_seq: truncates sequences bigger
        #                   than this
        max_imgs_per_seq: 4
        # CAUTION: If using concat combine mode,
        # min_imgs_per_seq must be equal to max_imgs_per_seq

        # indicates if padding should be added before or
        # after the original sequence
        # supported modes: post, pre
        sequence_padding: post
        # indicates if sequences longer than the maximum
        # size should be truncated from the start or the end
        # supported modes: post, pre
        sequence_truncating: post

# Configuration during training
train_config:
    epochs: 30         # epoch = whole training dataset
    batch_size: 16
    train_split: 0.9
    val_split: 0.1
    seed: 1111         # seed the random generators
    shuffle: true      # shuffle dataset between epochs

    optimizer: adam
    learning_rate: 1.0e-04
    epsilon:  1.0e-08
    decay: 0.0

    loss: 'categorical_crossentropy'
    # metrics: List of metrics to collect during training
    #          and validation
    # Supports:
    #   acc: train accuracy
    #   auc: area under the receiving operator curve
    metrics: [acc, auc]

    # stop_early: Metric whose progress to track to
    #             perform early stopping
    #          Supports:
    #             loss: training loss
```

```
    #             val_loss: validation loss
    #             acc: training accuracy
    #             val_acc: validation accuracy
    #             null: disable stop early
    # min_delta: minimum change in target metric that
    #            registers as an improvement
    # patience: number of epochs without an improvement
    #           before stopping early
    stop_early: loss
    min_delta: 0
    patience: 3

    # class_weight: if true, example losses are scaled to
    #               give more importance to examples from
    #               underrepresented classes
    class_weight: false

    # fit_batch_first: if true, fits the model to a
    #                  single batch before training
    fit_batch_first: false

    # save_model: if false, the trained weights are
    #             discarded instead of saved
    save_model: false

model_config:
    # input_shape:
    #   for combine mode concat,
    #   use [seq_length, img_width, img_height, n_channels]
    #   for combine mode attention or last,
    #   use [null, img_width, img_height, n_channels]
    #   for single_tel mode ,
    #   use [img_width, img_height, n_channels]
    input_shape: [4, 108, 108, 1]

    # num_classes: how many classes to classify
    num_classes: 2

    # activation_function
    activation_function: 'relu'
    # dropout_rate: fraction of units to randomly drop
    #               between layers during training
    dropout_rate: 0.0
    # use_batchnorm: if true, adds batchnorm layers
    #                between convolutions
    use_batchnorm: true
    # l2_regularization: regularization penalty to the
    #                    kernels and biases
    l2_regularization: 0.0

    # cnn_layers: Conv2D layers
```

```
cnn_layers:
    - {filters: 32,  kernel_size: 3, use_maxpool: true}
    - {filters: 32,  kernel_size: 3, use_maxpool: true}
    - {filters: 64,  kernel_size: 3, use_maxpool: true}
    - {filters: 128, kernel_size: 3, use_maxpool: true}

# lstm_units: dimensionality of output of each LSTM cell
#             if null or 0 the LSTM is skipped
lstm_units: 2048
# combine_mode : specifies how the encoding of a
#                sequence is to be combined.
# Supports:
#   concat : outputs are stacked on top of one another
#   last : only last hidden state is returned
#   attention : combination via attention
combine_mode: attention

# fcn_layers: Dense layers
fcn_layers:
    - {units: 1024}
    - {units: 512}

# the model is finally fed to a dense layer with
# as many units as classes and a softmax activation
```

2.3 Implementation

The core of the program are several interdependent utility scripts, each of which can be run over the same .yaml configuration file to perform different functions.

In figure 2.1 we can see an UML diagram specifying the different components of the framework and how they relate to each other.

The most important scripts are:

- `train.py` : the training script coordinates the training process of a model, which can either be provided as an h5 file or provided in the model_config section of the yaml configuration file.
- `build_model.py` : this script contains the logic to build customizable CNN-RNN-FCN models from a yaml configuration file.
- `summarize.py` : this script produces useful graphical reports from the information generated during simple and grid search training.
- data package : these classes are responsible for loading the simulated data stored in .h5 format. Its main interface is the DataLoader class provided in the `data_loading.py` module. Other important modules in this package are `image_mapping.py` for mapping IACTs hexagonal grids to square arrays suited for CNNs and `data_processing.py` for preprocessing the sequences of images.

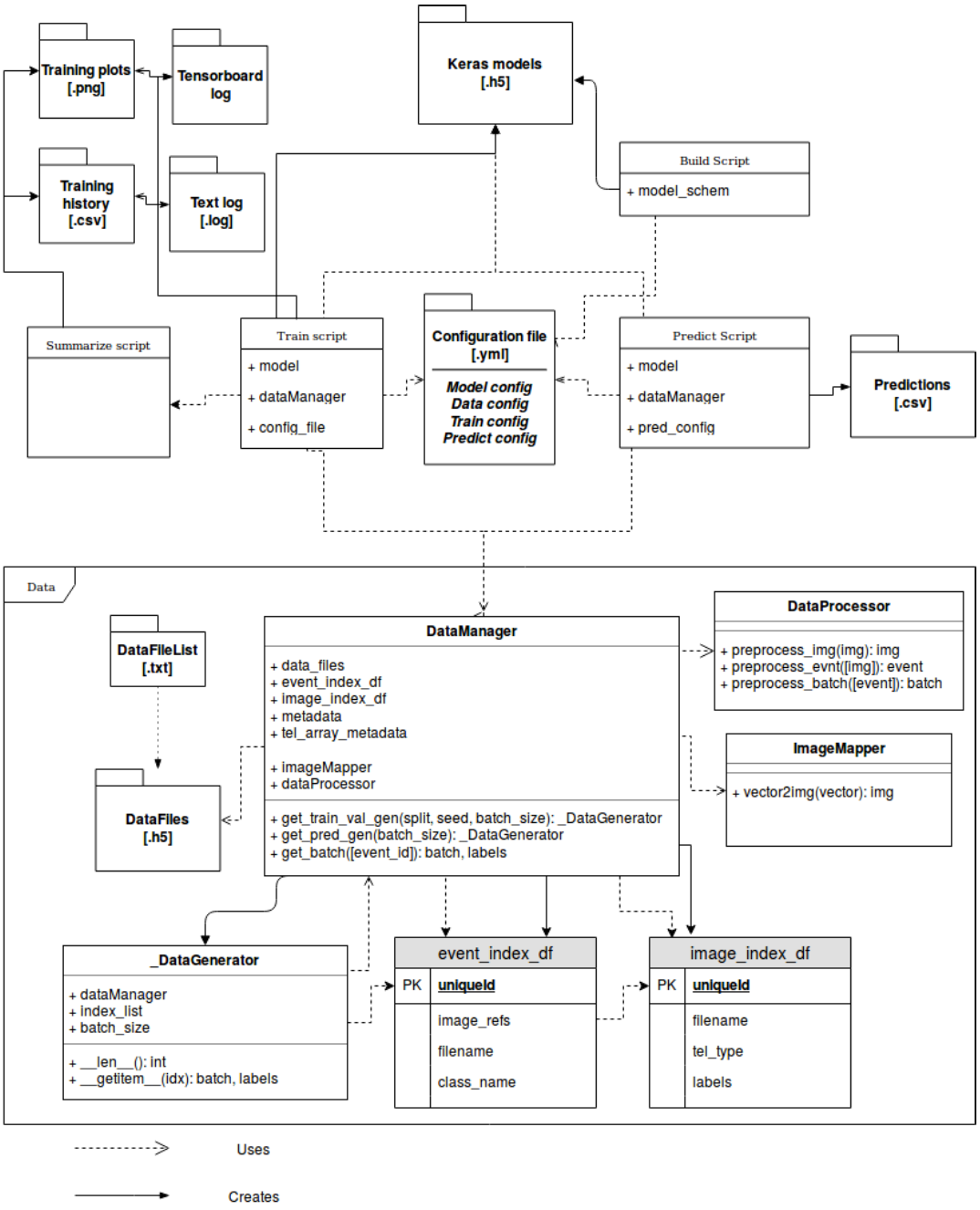


FIGURE 2.1: UML diagram describing the implementation of the Keras version of CTALearn

2.3.1 Feeding the data

In the .h5 files the charge per pixel data is stored in single row vectors. The image mapping module maps this vector to a 2D array representing how the image would be if the pixels of the telescopes were square instead of hexagonal.

Each data point that will be fed to the network is an array of images (an "event") together with a label indicating whether the event originated from a gamma ray or a proton simulation.

We can choose multiple parameters to regulate how this data is presented. For example we can change the size of the resulting 2D arrays, we can change how the images inside an event will be ordered and we can regulate the number of images per event.

Within an event we have chosen to order the images in increasing total brightness. Thus the brighter and thus, we can suppose, more relevant images will be at the end, closer to the final output of the image in the LSTM model (in the attention model, the order is irrelevant).

For all our experiments, we will be using between 1 and 8 images per event. In order to allow batching together events of different sequence length, we will left-pad sequences with blank images when needed.

We remark that even though our models will be in principle capable of processing arbitrarily many images per event, in practice we have chosen an upper limit of images per event. This allows us to compare our results to the v0.2.0 CTLearn benchmark in fair conditions, as all models will have seen the same quantity of images during training, and prevents us from running out of GPU memory and extending the training time for too long.

2.3.2 Grid Search

A key functionality provided by the `train.py` script is **grid search**.

Deep Learning models and training sessions incorporate several hyperparameters of paramount importance for performance, such as the learning rate or the number of units per layer. The output of the model does not differentially depend on those hyperparameters, and thus we cannot use gradient descent to optimize them.

Grid search enables a systematic exploration of how changing the hyperparameters influences the output of the model, trying every possible combination of the hyperparameter values we have specified.

To enable grid search in a model, append the prefix `multi\_` to the configuration options you want to search, and change their values for a list of values indicating the search range.

2.4 Dependencies and installation

The final framework has the following dependencies:

```
name: ctalearn
channels:
  - defaults
dependencies:
  - _tflow_select=2.1.0=gpu
  - absl-py=0.7.1=py37_0
  - astor=0.7.1=py37_0
  - astropy=3.1.2=py37h7b6447c_0
  - atomicwrites=1.3.0=py37_1
  - attrs=19.1.0=py37_1
  - blas=1.0=mkl
  - blosc=1.15.0=hd408876_0
  - bzip2=1.0.6=h14c3975_5
  - c-ares=1.15.0=h7b6447c_1
  - ca-certificates=2019.1.23=0
  - cairo=1.14.12=h8948797_3
  - certifi=2019.3.9=py37_0
  - cloudpickle=0.8.1=py_0
  - cudatoolkit=10.0.130=0
  - cudnn=7.3.1=cuda10.0_0
  - cupti=10.0.130=0
  - cycler=0.10.0=py37_0
  - cytoolz=0.9.0.1=py37h14c3975_1
  - dask-core=1.2.1=py_0
  - dbus=1.13.6=h746ee38_0
  - decorator=4.4.0=py37_1
  - expat=2.2.6=he6710b0_0
  - ffmpeg=4.0=hcdf2ecd_0
  - fontconfig=2.13.0=h9420a91_0
  - freeglut=3.0.0=hf484d3e_5
  - freetype=2.9.1=h8a8886c_1
  - gast=0.2.2=py37_0
  - glib=2.56.2=hd408876_0
  - graphite2=1.3.13=h23475e2_0
  - grpcio=1.16.1=py37hf8bcb03_1
  - gst-plugins-base=1.14.0=hbbd80ab_1
  - gstreamer=1.14.0=hb453b48_1
  - h5py=2.8.0=py37h989c5e5_3
  - harfbuzz=1.8.8=hffaf4a1_0
  - hdf5=1.10.2=hba1933b_1
  - icu=58.2=h9c2bf20_1
  - imageio=2.5.0=py37_0
  - intel-openmp=2019.3=199
  - jasper=2.0.14=h07fcdf6_1
  - jpeg=9b=h024ee3a_2
  - keras-applications=1.0.7=py_0
  - keras-preprocessing=1.0.9=py_0
  - kiwisolver=1.1.0=py37he6710b0_0
  - libedit=3.1.20181209=hc058e9b_0
  - libffi=3.2.1=hd88cf55_4
  - libgcc-ng=8.2.0=hdf63c60_1
  - libgfortran-ng=7.3.0=hdf63c60_0
  - libglu=9.0.0=hf484d3e_1
  - libopencv=3.4.2=hb342d67_1
  - libopus=1.3=h7b6447c_0
  - libpng=1.6.37=hbc83047_0
  - libprotobuf=3.7.1=hd408876_0
  - libstdcxx-ng=8.2.0=hdf63c60_1
  - libtiff=4.0.10=h2733197_2
  - libuuid=1.0.3=h1bed415_2
  - libvpx=1.7.0=h439df22_0
  - libxcb=1.13=h1bed415_1
  - libxml2=2.9.9=he19cac6_0
  - lzo=2.10=h49e0be7_2
  - markdown=3.1=py37_0
  - matplotlib=3.0.3=py37h5429711_0
  - mkl=2019.3=199
  - mkl_fft=1.0.12=py37ha843d7b_0
  - mkl_random=1.0.2=py37hd81dba3_0
  - mock=2.0.0=py37_0
  - more-itertools=7.0.0=py37_0
  - ncurses=6.1=he6710b0_1
  - networkx=2.3=py_0
  - numexpr=2.6.9=py37h9e4a6bb_0
  - numpy=1.16.3=py37h7e9f1db_0
  - numpy-base=1.16.3=py37hde5b4d6_0
  - olefile=0.46=py37_0
  - opencv=3.4.2=py37h6fd60c2_1
  - openssl=1.1.1b=h7b6447c_1
  - pandas=0.24.2=py37he6710b0_0
  - pbr=5.1.3=py_0
  - pcre=8.43=he6710b0_0
  - pillow=6.0.0=py37h34e0f95_0
  - pip=19.1=py37_0
  - pixman=0.38.0=h7b6447c_0
  - pluggy=0.9.0=py37_0
  - protobuf=3.7.1=py37he6710b0_0
  - psutil=5.6.2=py37h7b6447c_0
  - py=1.8.0=py37_0
  - py-opencv=3.4.2=py37hb342d67_1
  - pyparsing=2.4.0=py_0
  - pyqt=5.9.2=py37h05f1152_2
```

```
- pytables=3.4.4=py37ha205bf6_0      - snappy=1.1.7=hbae5bb6_3
- pytest=4.4.1=py37_0                - sqlite=3.28.0=h7b6447c_0
- pytest-arraydiff=0.3=py37h39e3cac_0- tensorboard=1.13.1=py37hf484d3e_0
- pytest-astropy=0.5.0=py37_0        - tensorflow=1.13.1=gpu_py37hc158e3b_0
- pytest-doctestplus=0.3.0=py37_0    - tensorflow-base=1.13.1=gpu_py37h8d69cac_0
- pytest-openfiles=0.3.2=py37_0      - tensorflow-estimator=1.13.0=py_0
- pytest-remotedata=0.3.1=py37_0     - tensorflow-gpu=1.13.1=h0d30ee6_0
- python=3.7.3=h0371630_0            - termcolor=1.1.0=py37_1
- python-dateutil=2.8.0=py37_0       - tk=8.6.8=hbc83047_0
- pytz=2019.1=py_0                   - toolz=0.9.0=py37_0
- pywavelets=1.0.3=py37hdd07704_1    - tornado=6.0.2=py37h7b6447c_0
- pyyaml=5.1=py37h7b6447c_0          - werkzeug=0.15.2=py_0
- qt=5.9.7=h5867ecd_1                - wheel=0.33.1=py37_0
- readline=7.0=h7b6447c_5            - xz=5.2.4=h14c3975_4
- scikit-image=0.15.0=py37he6710b0_0 - yaml=0.1.7=had09818_2
- scipy=1.2.1=py37h7c811a0_0         - zlib=1.2.11=h7b6447c_3
- setuptools=41.0.1=py37_0           - zstd=1.3.7=h0b5b093_0
- sip=4.19.8=py37hf484d3e_0          - pip:
- six=1.12.0=py37_0                    - ctalearn==0.1
```

To install our framework, first clone the repository and run

```
conda create -n ctalearn_env --file environment.yml
```

where `environment.yml` is a text file containing the previous dependencies. This will create a conda environment with all dependencies installed.

Activate the environment with the command

```
conda activate ctalearn_env
```

At this point the installation will be complete.

Now you can write your own YAML configuration file, following the scheme of the example configuration file above.

Afterwards, run your own experiment using the command

```
python ctalearn/train my_training_config.yaml
```

After invoking the program, you should see some information come up on the console, eventually showing the dataset metadata and finally the keras training bar. Once the training is complete, in the folder where the command was executed you will find the output of the run, including training plots, a csv summary, a folder with the tensorboard log and a textual summary of the model, plus the trained model in h5 format if you chose the `save_model:true option`.

If you wish to run a grid search, append the relevant fields in the configuration with the prefix `multi_` and indicate the hyperparameter range with a list.

If you change code in the package, remember to execute `pip install .` from the root folder of the repo to make the changes effective before launching a run.

Chapter 3

LSTM Model

3.1 LSTM Architecture

As discussed in the introduction, we plan to implement an architecture capable of handling as input sets of images of non fixed cardinal. In this section we discuss the details of this architecture.

Our architecture will feature three parts: a CNN, a LSTM layer and a FCN. See a visual representation of the model in 3.1.

3.1.1 Dynamic unrolling

Since our process does not need the outcome of every LSTM cell and only of the last one, we can perform **dynamic unrolling** to lessen the memory usage of the network during both training and inference.

To do this we only load in memory one cell simultaneously, while CTLearn's CNN-RNN model was forced to statically load one cell per each sequence element in order to concatenate their outputs.

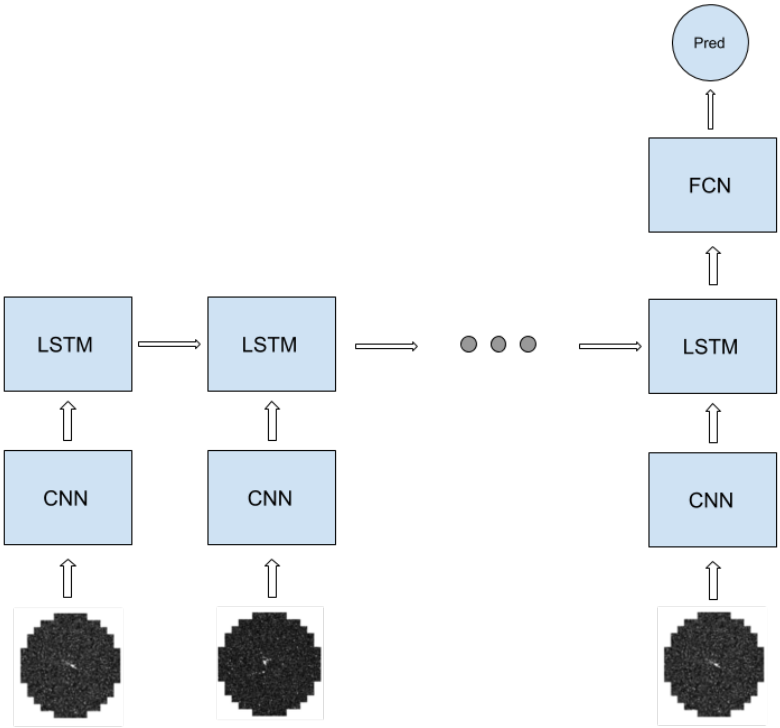


FIGURE 3.1: LSTM model. Image features are fed to successive LSTM units, and the last LSTM unit produces a embedding of the image ensemble over which the classification is made. The sample input images have been taken from [Nie+17]

3.1.2 Final architecture summary

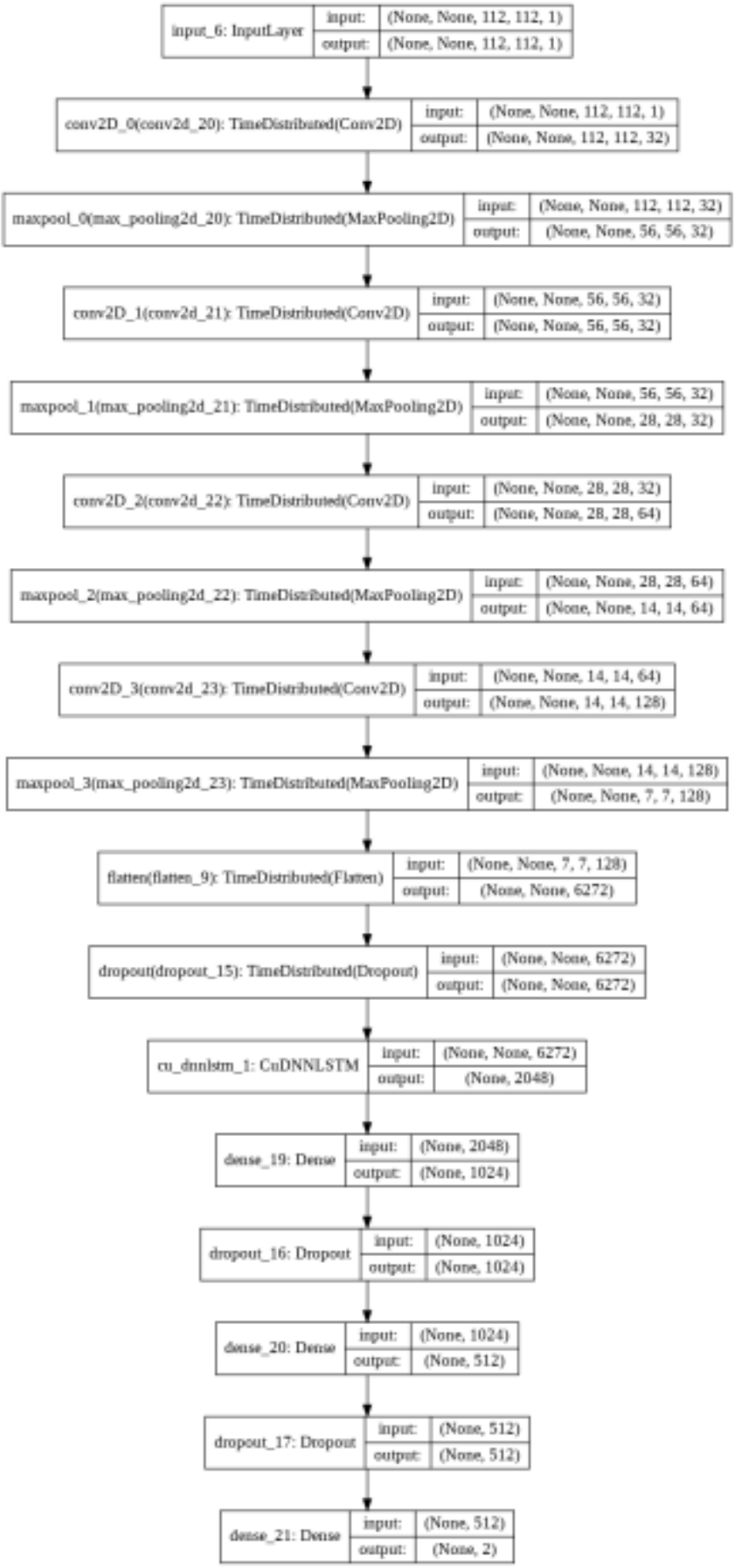


FIGURE 3.2: Visualization of the LSTM model in Keras

3.2 Experimental set up

To tune the hyperparameters of the LSTM network we will use a iterated grid search procedure.

First phase will attempt to get the best possible training score we can manage on the training set, using data from a single telescope. Second phase will refine the results of the first phase using regularization mechanisms.

Once the second phase is finished, the best performing model overall will be trained on data from the different telescopes

3.2.1 Initial search

The first step in our experiments was running a coarse grid search with data from a single telescope, the MSTF.

The training configuration used for the initial grid search used was the following:

```
data_config:
    file_list_fn: data.txt
    channels: [image_charge]
    img_size: [112, 112]
    selected_tel_types: ['MSTF']
    data_type: 'array'
    min_triggers_per_event: 1
    image_mapping_config:
        hex_conversion_algorithm:
            oversampling
    preprocessing_config:
        normalization: null
        resize_mode: 'interpolate'
        event_order_type: size
        event_order_reverse: false
        min_imgs_per_seq: 1
        max_imgs_per_seq: 8
        sequence_padding: pre
        sequence_truncating: post
train_config:
    epochs: 10
    batch_size: 16
    train_split: 0.9
    val_split: 0.1
    seed: 1111
    shuffle: true
    optimizer: adam
    multi_learning_rate:
        [1.0e-04, 1.0e-05]
    epsilon:  1.0e-08
    decay: 0.0
    loss: 'categorical_crossentropy'
    metrics: [acc, auc]
    stop_early: loss
    min_delta: 0
    patience: 3
    class_weight: false
    multi_fit_batch_first:
        [false, true]
    save_model: false
model_config:
    input_shape: [null, 112, 112, 1]
    num_classes: 2
    activation_function: 'relu'
    dropout_rate: 0.0
    multi_use_batchnorm:
        [false, true]
    l2_regularization: 0.0
    cnn_layers:
      - filters: 32
        kernel_size: 3
        use_maxpool: true
      - filters: 32
        kernel_size: 3
        use_maxpool: true
      - filters: 64
        kernel_size: 3
        use_maxpool: true
      - filters: 128
        kernel_size: 3
        use_maxpool: true
    lstm_units: 2048
    combine_mode: last
    fcn_layers:
        - {units: 1024}
        - {units: 512}
```

Our goal is to try overfitting the model to the training data, so we are not adding any regularization features, and the stop early mechanism is tracking the training loss.

The previous configuration launches 8 runs total. The differences in hyperparameters are compilated in table 3.1.

Run Number	Learning Rate	Fit first	Batchnorm
000	1e-4	False	False
001	1e-4	False	True
002	1e-4	True	False
003	1e-4	True	True
004	1e-5	False	False
005	1e-5	False	True
006	1e-5	True	False
007	1e-5	True	True

TABLE 3.1: Distinguishing hyperparameters of each run of the first stage of the LSTM grid search

The learning rate is the initial magnitude of change of the parameters for each batch processed.

Fit first indicates whether we start training directly with the whole training set (if it is False) or if we first try to overfit the model on a single batch (if it is True).

Batchnorm indicates whether we use Batchnorm layers in the neural network. Batchnorm layers adjust the activations to have mean 0 and std 1.

After running these settings, we found the results summarized in figure 3.3.

Let us focus on the accuracy metric for now, looking at the accuracy plots for each of the runs in figure 3.4.

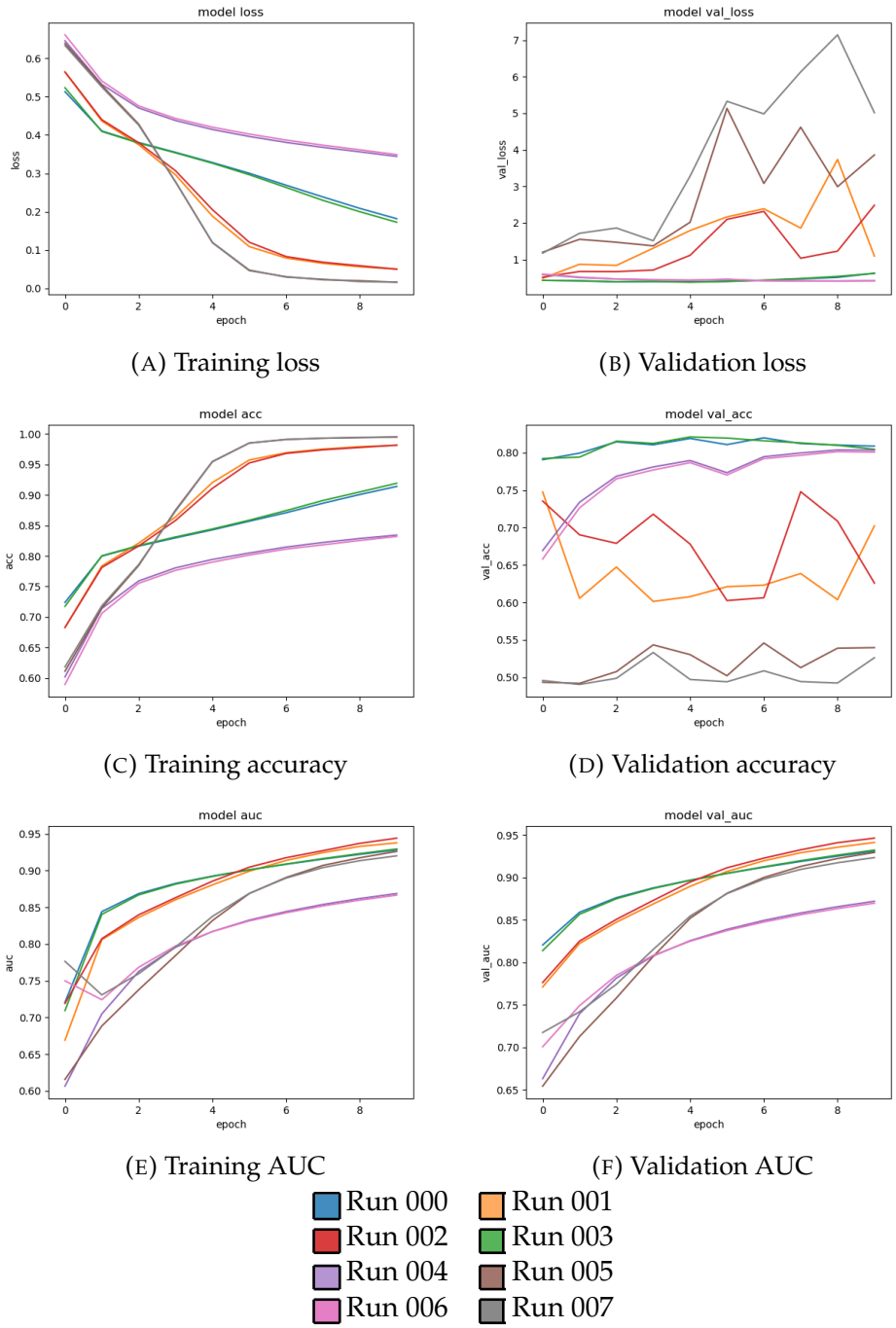
We notice that fitting first to a single batch makes almost no difference in the final result. Compare for example the training plot of run 005 and run 007.

A faster learning rate improves the accuracy score on the training set while not using BatchNorm, but leads to no improvements in the validation accuracy.

While using BatchNorm (odd run numbers) we get extremely good results in the training set, but really disappointing results in validation, where we get between 0.5 and 0.6 accuracy.

One plausible interpretation of this behaviour is that BatchNorm speeds up the training considerably, and leads to overfitting immediately.

In any case, run 000 obtained the best results overall in validation accuracy, and we will be using it as a base for the next experiment.



(A) Training loss

(B) Validation loss

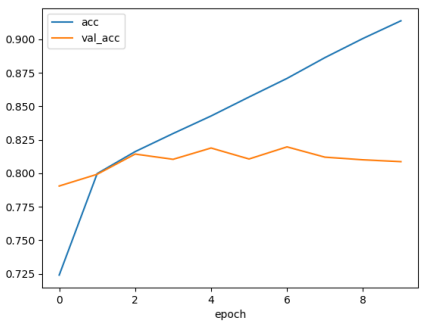
(C) Training accuracy

(D) Validation accuracy

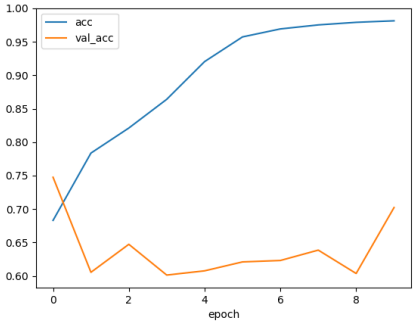
(E) Training AUC

(F) Validation AUC

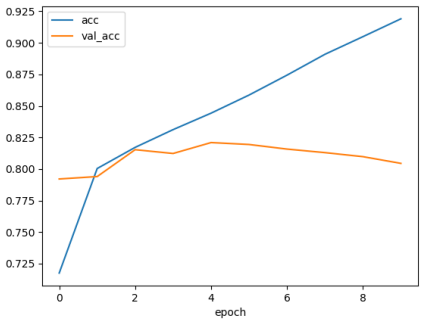
FIGURE 3.3: Summarized results of first stage of the LSTM grid search



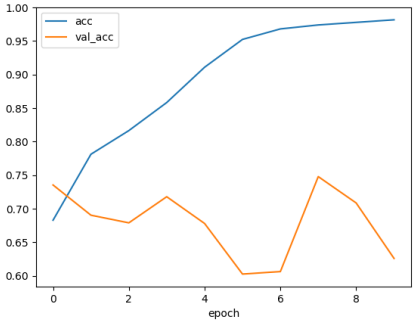
(A) Run 000



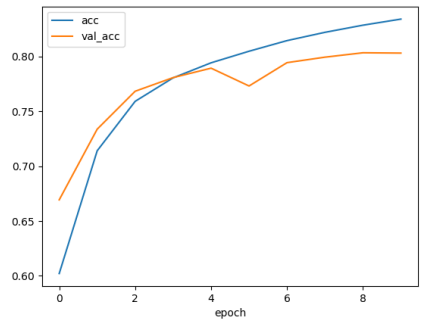
(B) Run 001



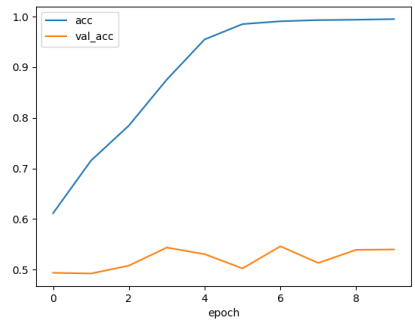
(C) Run 002



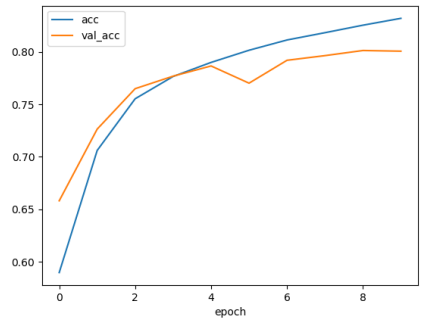
(D) Run 003



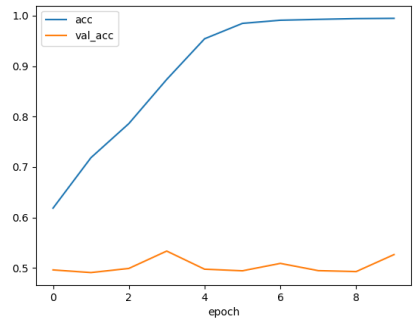
(E) Run 004



(F) Run 005



(G) Run 006



(H) Run 007

FIGURE 3.4: Accuracy training plots of each run of the first stage of the LSTM grid search

3.2.2 Second iteration

After the coarse grid search, we selected the best performing model and run a smaller grid search with parameters similar to those of the best run.

The training configuration we used is the following:

```
data_config:
    file_list_fn: data.txt

    channels: [image_charge]

    img_size: [112, 112]

    selected_tel_types: ['MSTF']

    data_type: 'array'

    min_triggers_per_event: 1

    image_mapping_config:
        hex_conversion_algorithm:
            oversampling

    preprocessing_config:
        normalization: null
        resize_mode: 'interpolate'

        event_order_type: size
        event_order_reverse: false

        min_imgs_per_seq: 1
        max_imgs_per_seq: 8

        sequence_padding: pre
        sequence_truncating: post

train_config:
    epochs: 10
    batch_size: 16
    train_split: 0.9
    val_split: 0.1
    seed: 1111
    shuffle: true

    optimizer: adam
    learning_rate: 1.0e-04
    epsilon:  1.0e-08
    decay: 0.0

    loss: 'categorical_crossentropy'
    metrics: [acc, auc]

    stop_early: val_loss
    min_delta: 0
    patience: 2

    class_weight: false

    fit_batch_first: false

    save_model: false

model_config:
    input_shape:
        [null, 112, 112, 1]
    num_classes: 2
    activation_function: 'relu'
    multi_dropout_rate:
        [0.0, 0.5, 0.8]
    use_batchnorm: false
    multi_l2_regularization:
        [0.0, 1.0e-04]

    cnn_layers:
      - filters: 32
        kernel_size: 3
        use_maxpool: true
      - filters: 32
        kernel_size: 3
        use_maxpool: true
      - filters: 64
        kernel_size: 3
        use_maxpool: true
      - filters: 128
        kernel_size: 3
        use_maxpool: true

    lstm_units: 2048
    combine_mode: last

    fcn_layers:
        - {units: 1024}
        - {units: 512}
```

Run Number	Dropout	L2 Weight
000	0.0	0.0
001	0.0	1e-04
002	0.5	0.0
003	0.5	1e-04
004	0.8	0.0
005	0.8	1e-04

TABLE 3.2: Distinguishing hyperparameters of each run of the second stage of the LSTM grid search

That is, the regularization mechanisms we are trying are early stopping tracking validation loss, L2 regularization and dropout.

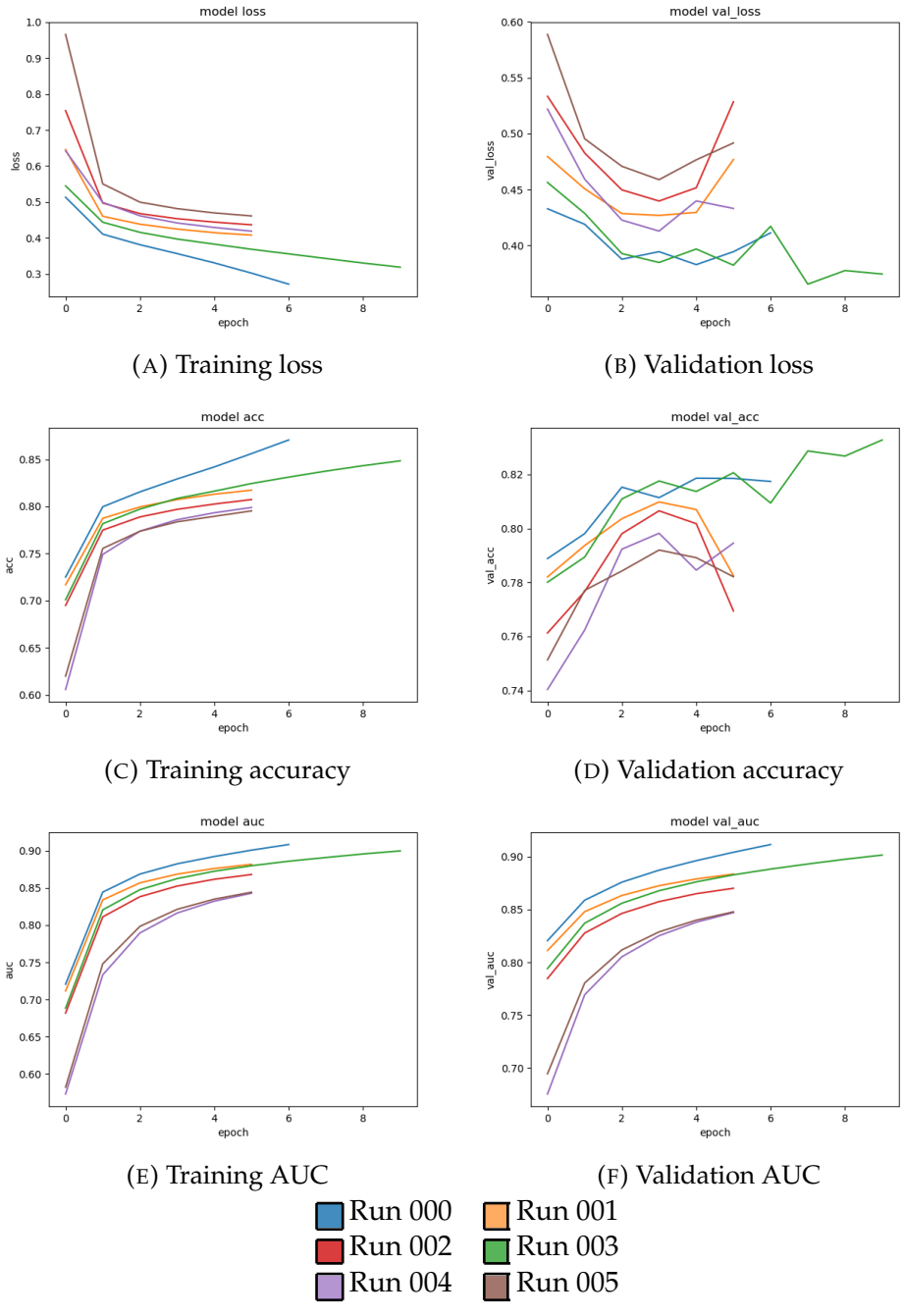
The defining differences between each run are listed in figure 3.2.

After the experiments finished we found the results summarized in figure 3.5.

We can appreciate better how the regularization mechanisms affected training by looking at the accuracy training plots of each run in figure 3.6.

Regularization has helped lessen the gap between the training and validation results, but we do not get much better absolute results in accuracy.

Best performance in validation has been achieved by using a small dropout rate, in run 002.



(A) Training loss

(B) Validation loss

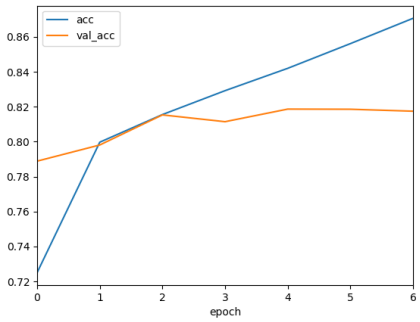
(C) Training accuracy

(D) Validation accuracy

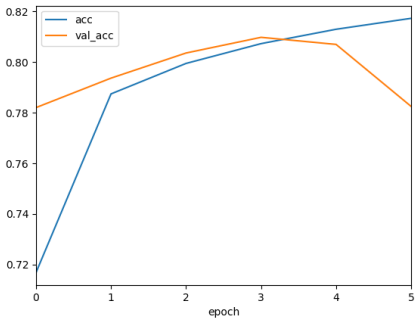
(E) Training AUC

(F) Validation AUC

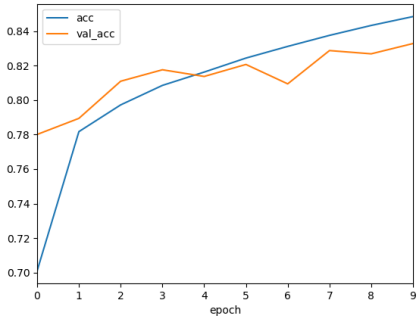
FIGURE 3.5: Summarized results of the first stage of the LSTM hyper-parameter grid search



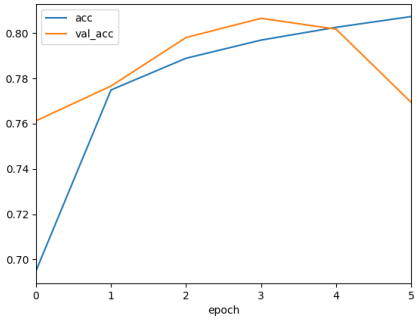
(A) Run 000



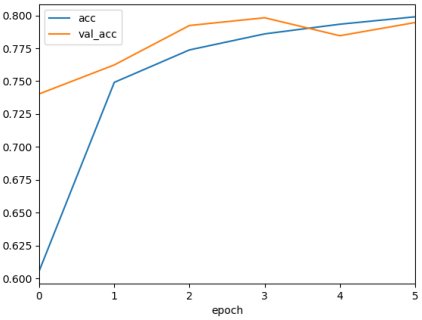
(B) Run 001



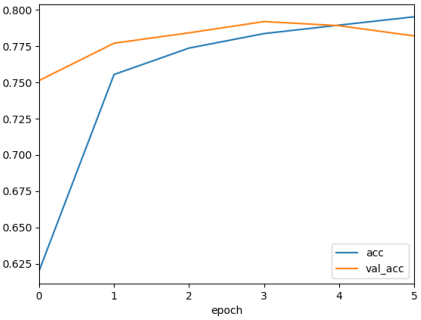
(C) Run 002



(D) Run 003



(E) Run 004



(F) Run 005

FIGURE 3.6: Accuracy training plots of each run of the first stage of the LSTM hyperparameter grid search

3.2.3 Benchmarking

Lastly we will benchmark the best performing hyperparameters we have found against all telescope types.

The configuration file used is reproduced next:

```
data_config:
  channels:
  - image_charge
  data_type: array
  file_list_fn: data.txt
  image_mapping_config:
    hex_conversion_algorithm:
      oversampling
  img_size: [112, 112]
  min_triggers_per_event: 1
  preprocessing_config:
    event_order_reverse: false
    event_order_type: size
    max_imgs_per_seq: 8
    min_imgs_per_seq: 1
    normalization: null
    resize_mode: interpolate
    sequence_padding: pre
    sequence_truncating: post
  multi_selected_tel_types:
    [[LST], [MSTF], [MSTN], [MSTS],
     [SSTC], [SST1], [SSTA]]
model_config:
  activation_function: relu
  cnn_layers:
  - filters: 32
    kernel_size: 3
    use_maxpool: true
  - filters: 32
    kernel_size: 3
    use_maxpool: true
  - filters: 64
    kernel_size: 3
    use_maxpool: true
  - filters: 128
    kernel_size: 3
    use_maxpool: true
  combine_mode: last
  dropout_rate: 0.5
  fcn_layers:
  - units: 1024
  - units: 512
  input_shape: [null, 112, 112, 1]
  l2_regularization: 0.0
  lstm_units: 2048
  num_classes: 2
  use_batchnorm: false
train_config:
  batch_size: 16
  class_weight: false
  decay: 0.0
  epochs: 10
  epsilon: 1.0e-08
  fit_batch_first: false
  learning_rate: 0.0001
  loss: categorical_crossentropy
  metrics: [acc, auc]
  min_delta: 0
  optimizer: adam
  patience: 2
  save_model: false
  seed: 1111
  shuffle: true
  stop_early: val_loss
  train_split: 0.9
  val_split: 0.1
```

The final results on validation are compiled in table 3.3.

Telescope type	Events	Positive	Negative	AUROC	Accuracy	Training time
LST	8541	3884	4656	0.8203	74.06%	0:41:12
MSTF	24982	12122	12859	0.8837	81.62%	1:32:50
MSTN	26937	13037	13899	0.9177	85.43%	2:07:06
MSTS	22306	11091	11214	0.9071	83.58%	2:28:27
SST1	19788	10337	9451	0.8906	80.88%	1:45:08
SSTA	18367	9445	8922	0.8782	81.25%	1:11:59
SSTC	19064	9789	9275	0.9157	84.59%	1:34:03

TABLE 3.3: Validation results for LSTM model, v0.2.0 benchmark

3.3 Multiple telescopes

Our new framework includes a functionality not present in the v0.2.0 CTLearn framework: training with data from multiple telescopes.

We have configured a run with this feature.

```
data_config:
  channels:
  - image_charge
  data_type: array
  file_list_fn: data.txt
  image_mapping_config:
    hex_conversion_algorithm:
        oversampling
  img_size: [112, 112]
  min_triggers_per_event: 1
  preprocessing_config:
    event_order_reverse: false
    event_order_type: size
    max_imgs_per_seq: 8
    min_imgs_per_seq: 1
    normalization: null
    resize_mode: interpolate
    sequence_padding: pre
    sequence_truncating: post
  selected_tel_types:
    [LST, MSTF, MSTN, MSTS,
     SSTC, SST1, SSTA]
train_config:
  batch_size: 16
  class_weight: false
  decay: 0.0
  epochs: 10
  epsilon: 1.0e-08
  fit_batch_first: false
  learning_rate: 0.0001
  loss: categorical_crossentropy
  metrics: [acc, auc]
  min_delta: 0
  optimizer: adam
  patience: 2
  save_model: false
  seed: 1111
  shuffle: true
  stop_early: val_loss
  train_split: 0.9
  val_split: 0.1
model_config:
  activation_function: relu
  cnn_layers:
  - filters: 32
    kernel_size: 3
    use_maxpool: true
  - filters: 32
    kernel_size: 3
    use_maxpool: true
  - filters: 64
    kernel_size: 3
    use_maxpool: true
  - filters: 128
    kernel_size: 3
    use_maxpool: true
  combine_mode: last
  dropout_rate: 0.5
  fcn_layers:
  - units: 1024
  - units: 512
  input_shape: [null, 112, 112, 1]
  l2_regularization: 0.0
  lstm_units: 2048
  num_classes: 2
  use_batchnorm: false
```

In this mode, for each event we feed to our model images from all telescopes, ordered by their brightness. For events with more than 8 images we only feed to the model the 8 brightest (albeit in principle we could feed it as many as we'd like).

See the loss, accuracy and AUROC training graphs in figure 3.7.

We appreciate that the model has overfitted the training set, looking at the loss and accuracy plots.

The results of the run are summarized in table 3.4.

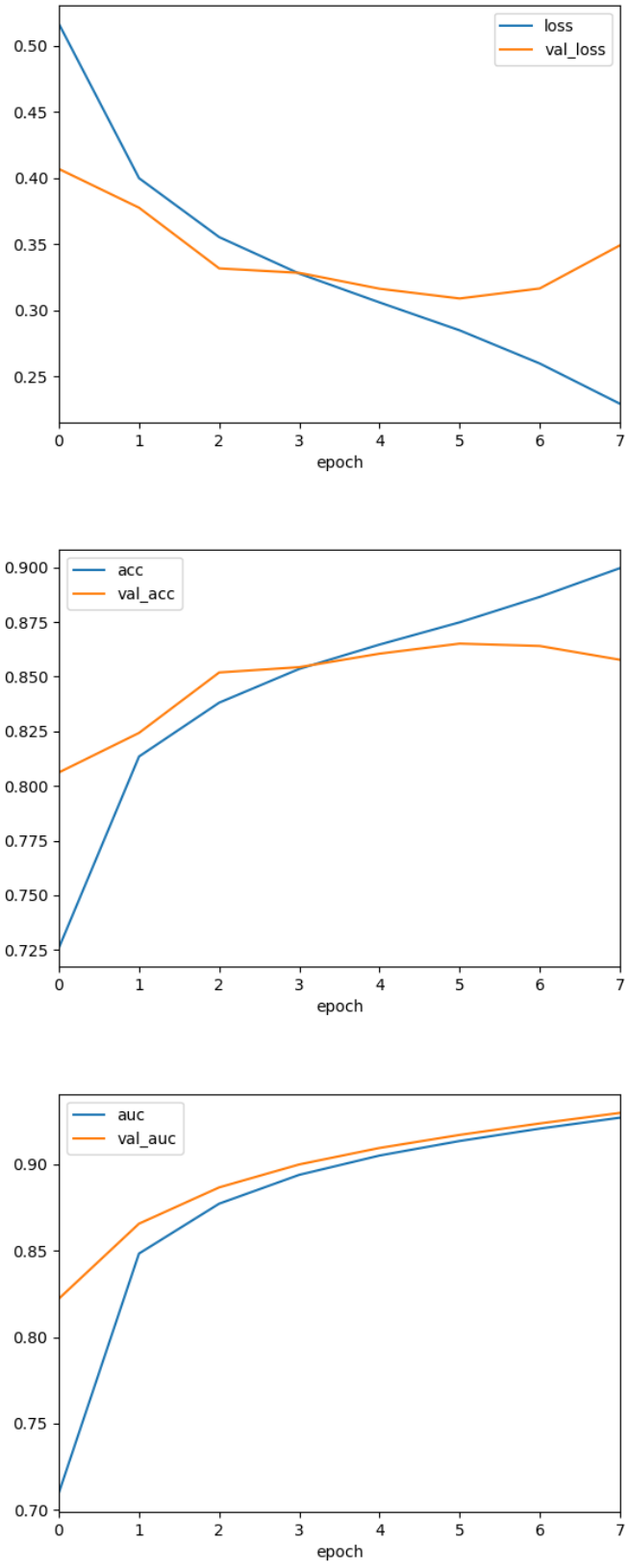


FIGURE 3.7: LSTM training plots when trained on all telescopes

Telescope type	Events	Positive	Negative	AUROC	Accuracy	Training time
ALL	37710	18941	18769	0.9298	85.77%	12:54:10

TABLE 3.4: Validation results for LSTM model trained on all telescopes

	CNN-RNN			LSTM		
Telescope type	AUROC	Accuracy	Training time	AUROC	Accuracy	Training time
LST	0.8285	73.43%	0:37:01	0.8203	74.06%	0:41:12
MSTF	0.8961	80.23%	2:06:32	0.8837	81.62%	1:32:50
MSTN	0.9169	83.10%	2:15:52	0.9177	85.43%	2:07:06
MSTS	0.9048	81.18%	2:37:34	0.9071	83.58%	2:28:27
SST1	0.8997	81.47%	3:21:48	0.8906	80.88%	1:45:08
SSTA	0.8556	75.27%	2:10:55	0.8782	81.25%	1:11:59
SSTC	0.9072	80.64%	1:51:05	0.9157	84.59%	1:34:03

TABLE 3.5: Validation results for LSTM model, v0.2.0 benchmark

3.4 LSTM experiments conclusion

We have trained a LSTM model via iterative grid search, and we have concluded that the best results are obtained with a moderate initial learning rate (1e-4), no batch-norm, some moderate dropout (0.5) and no L2 regularization. Furthermore we have learned that fitting a single batch first does not make any significant difference.

If we compare the results of our LSTM model with CTLearn's CNN-RNN model (see table 3.5) we realize that we are getting fairly similar results.

Most notable exceptions are the results over the SSTA and SSTC, where the LSTM model is able to outperform significantly the CNN-RNN baseline.

On the other cases we can observe that we are getting slightly better results on accuracy but slightly worse results on AUC, which makes sense given that we used accuracy as our optimization target metric during the grid search rather than AUROC.

(we should note that the training time is not directly comparable between benchmarks as the experiments were performed in different machines)

Finally, we notice that using data from all telescopes results in vastly better accuracy performance, from the 83.15% accuracy using MSTN data to 85.77% accuracy in the all telescopes mode, at the cost of a vastly increased training time.

This points in the direction of ensemble models using data from many telescopes being key for effective classification.

Chapter 4

Attention Model

4.1 Attention model architecture

Here we describe the mathematical ideas behind the attention model.

This model has been inspired by the work in [Vas+17], where attention mechanisms are examined as a viable alternative to RNN schemes for sequence processing.

Attention mechanisms help neural networks focus on the most relevant parts of the input; in our case, it will help us give more importance to the input images that have more information about the photonic event captured by the CTA.

The concrete idea is the following: we start extracting features $v_1, ..., v_n$ from the images using a CNN network as before, but instead of feeding them to a LSTM cell we run a FCN network (the Scorer) on each of them, whose output is a single scalar number (its **attention score**).

Attention scores are normalized using a softmax so they sum to one and are between 0 and 1, and then we compute a weighted average of the features $v_1, ..., v_n$ extracted by the CNN using the normalized attention scores as weights.

$$Attention(v_1, ..., v_n) = Softmax(Scorer(v_1), ..., Scorer(v_n))^T (v_1, ..., v_n)$$

The result is fed to a FCN classifier that produces the final class prediction.

See figure 4.1 for a visual summary of the model.

4.1.1 Theoretical comparison of the LSTM and Attention model

The main theoretical advantage of the attention model over the LSTM model is that it significantly shortens the long range dependencies between the first images of the sequences and the final output, facilitating training.

As a tradeoff, the attention model is far more memory-intensive in comparison to the LSTM one, as it cannot be dynamically unrolled.

Another key difference is that the Attention architecture is order-independent, while the LSTM one is not and can be, in theory, greatly affected by ordering the input images according to a different criteria.

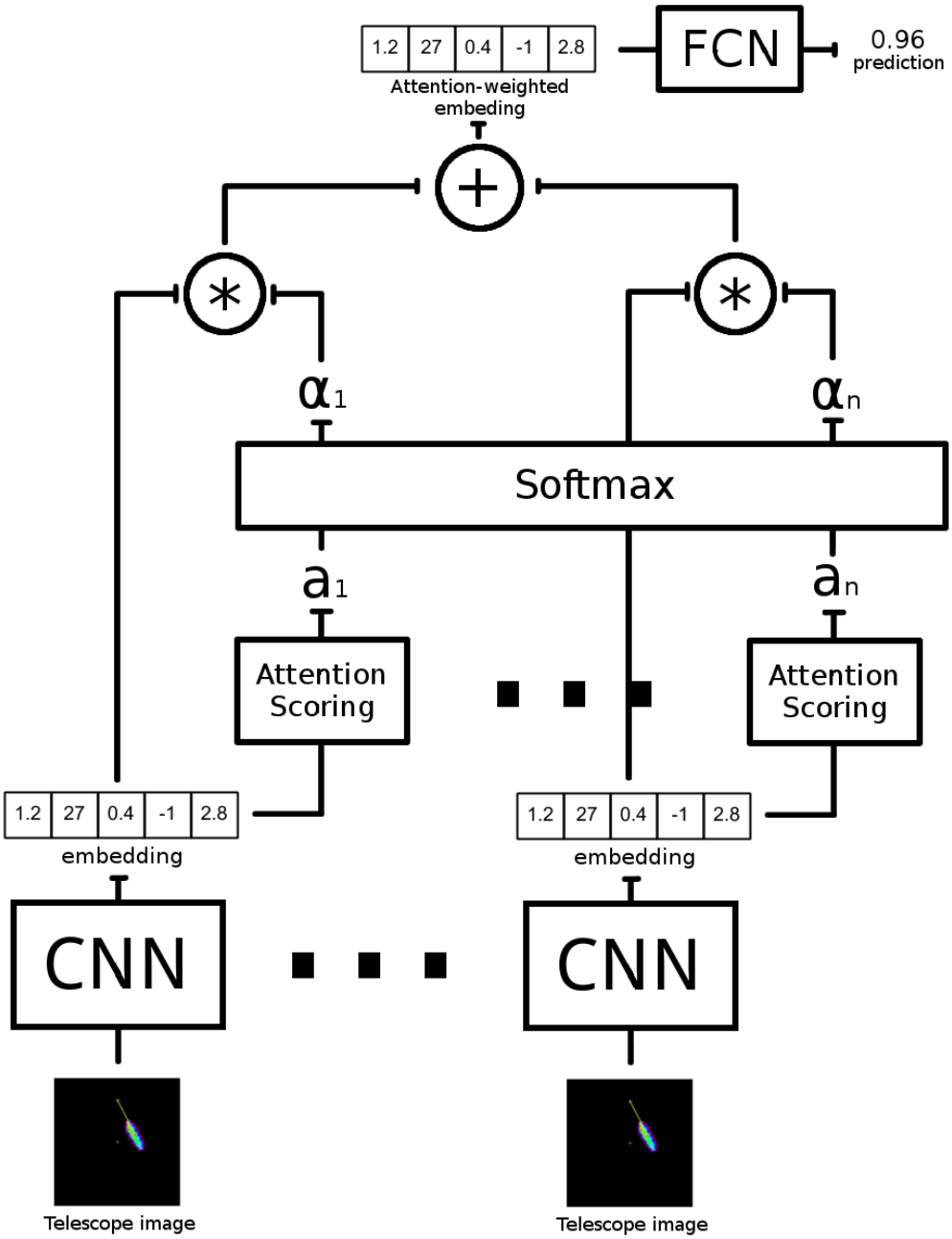


FIGURE 4.1: Our attention-based model. The feature vectors from each image are scored and combined via a weighted average.

4.1.2 Final architecture summary

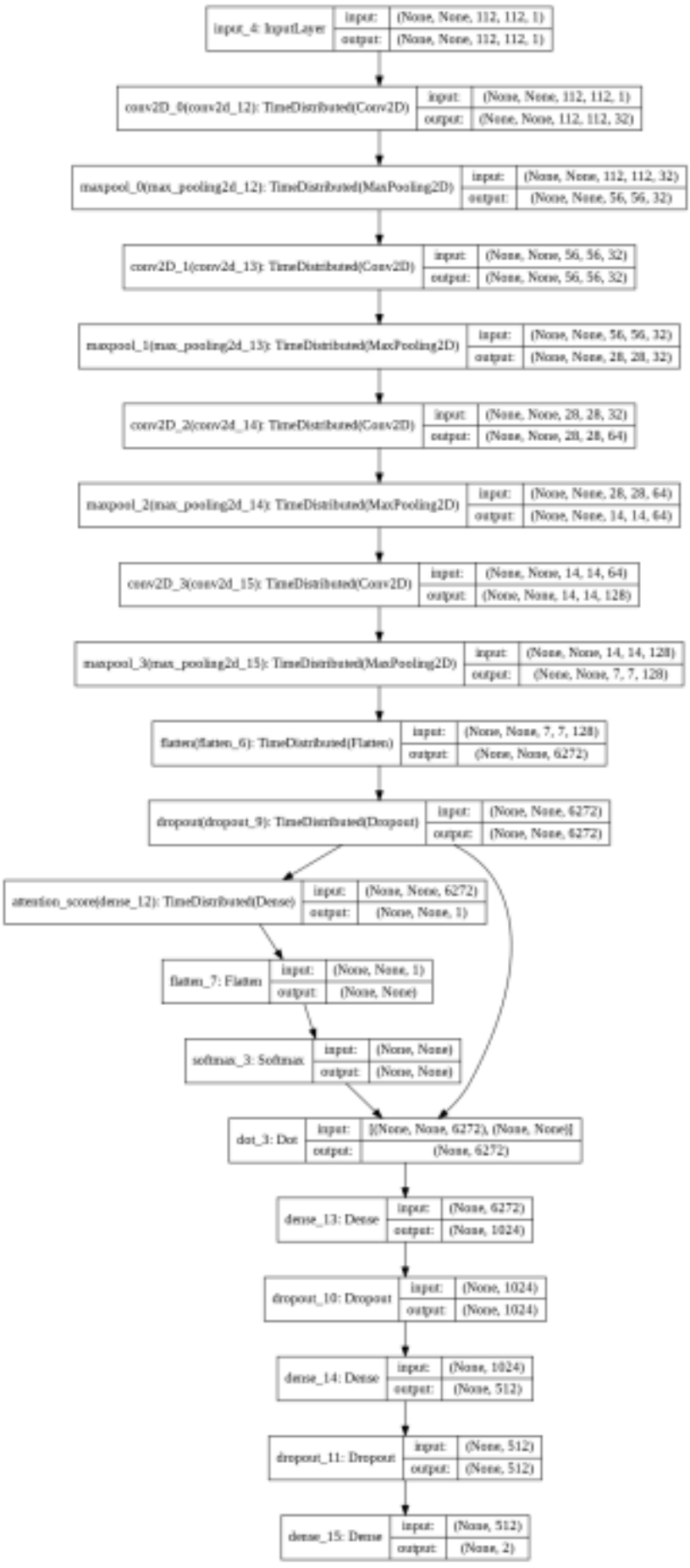


FIGURE 4.2: Visualization of the attention model in Keras

4.2 Experimental set up

To train the data we have used an iterative grid search process, where we first tried fitting our dataset to the training data and then adding regularization to improve the score in the validation set. Both of these have been done using data from the MSTF telescope.

The best performing hyperparameters used in the two first phases have been used to train one model per each type of telescope data to compare it to the CTLearn v0.2.0 benchmark. Finally we have trained one model using data from all telescopes.

4.2.1 Initial search

For our initial run we have opted to use the following grid search configuration:

```
data_config:
    file_list_fn: data.txt
    channels: [image_charge]
    img_size: [112, 112]
    selected_tel_types: ['MSTF']
    data_type: 'array'
    min_triggers_per_event: 1
    image_mapping_config:
        hex_conversion_algorithm:
            oversampling
    preprocessing_config:
        normalization: null
        resize_mode: 'interpolate'
        event_order_type: size
        event_order_reverse: false
        min_imgs_per_seq: 1
        max_imgs_per_seq: 8
        sequence_padding: pre
        sequence_truncating: post
    train_config:
    epochs: 10
    batch_size: 16
    train_split: 0.9
    val_split: 0.1
    seed: 1111
    shuffle: true
    optimizer: adam
    multi_learning_rate:
        [1.0e-04, 1.0e-05]
    epsilon:  1.0e-08
    decay: 0.0
    loss: 'categorical_crossentropy'
    metrics: [acc, auc]
    stop_early: loss
        min_delta: 0
        patience: 3
        class_weight: false
        multi_fit_batch_first:
            [false, true]
        save_model: false
model_config:
        input_shape:
            [null, 112, 112, 1]
        num_classes: 2
        activation_function: 'relu'
        dropout_rate: 0.0
        multi_use_batchnorm:
            [false, true]
        l2_regularization: 0.0
        cnn_layers:
            - filters: 32
              kernel_size: 3
              use_maxpool: true
            - filters: 32
              kernel_size: 3
              use_maxpool: true
            - filters: 64
              kernel_size: 3
              use_maxpool: true
            - filters: 128
              kernel_size: 3
              use_maxpool: true
        lstm_units: null
        combine_mode: attention
        fcn_layers:
            - {units: 1024}
            - {units: 512}
```

Run Number	Learning Rate	Fit first	Batchnorm
000	1e-4	False	False
001	1e-4	False	True
002	1e-4	True	False
003	1e-4	True	True
004	1e-5	False	False
005	1e-5	False	True
006	1e-5	True	False
007	1e-5	True	True

TABLE 4.1: Distinguishing hyperparameters of each run of the first stage of the Attention grid search

As with the LSTM, during the hyperparameter search we have opted to just run it using data from a single telescope, the MSTF.

Our goal is to try overfitting the model to the training data, so we are not adding any regularization features.

The configuration produces 8 runs, whose defining characteristics are summarized in table 4.1.

The results of the runs are summarized in figure 4.3.

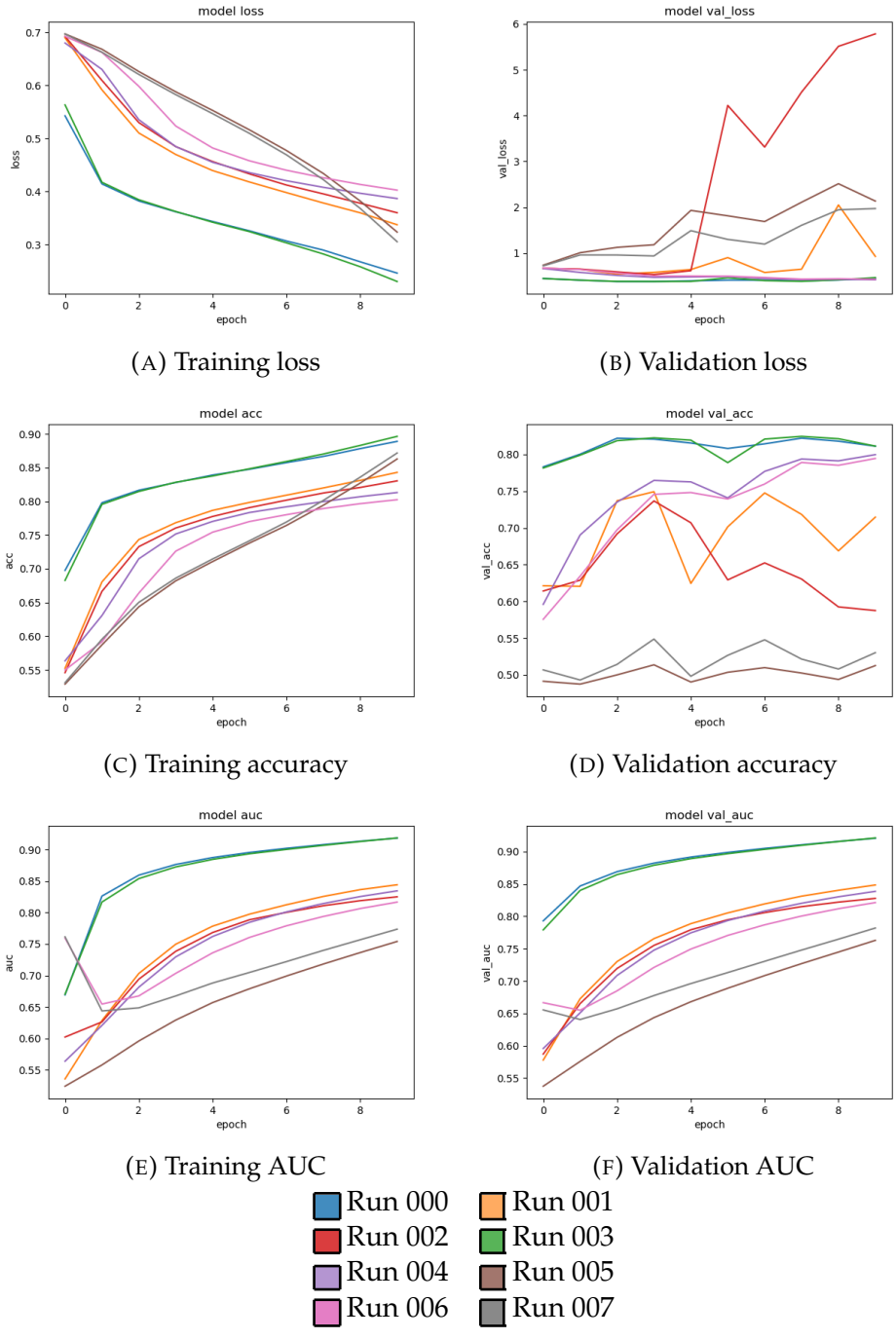
Let us focus on the accuracy metric for now, looking at the accuracy plots for each of the runs in table 4.4.

Once more we see that fitting first to a single batch makes no noticeable difference (compare run 000 to run 002 or 004 to 006).

Batchnorm again causes a variance problem, with a great gap between training and validation accuracy. However we note that this does not correspond entirely to overfitting, since in the runs with a large learning rate it actually caused the training performance to be worse overall (compare run 000 to run 001 or run 002 to run 003).

Once again, a bigger initial learning rate leads to better results.

In summary, we get very good results with a large initial learning rate and no batchnorm (run 000 and run 001). The accuracy graphs show a significant gap between validation a training performance, that we will train to address in the next iteration of the iterative grid search.



(A) Training loss

(B) Validation loss

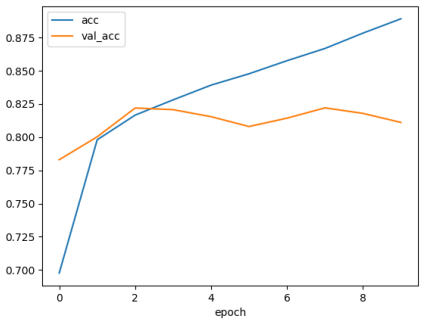
(C) Training accuracy

(D) Validation accuracy

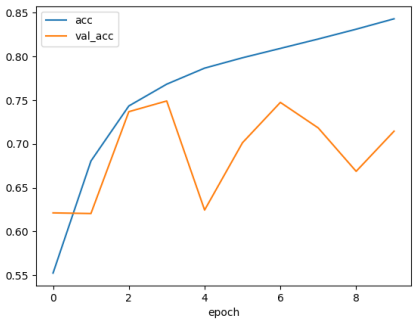
(E) Training AUC

(F) Validation AUC

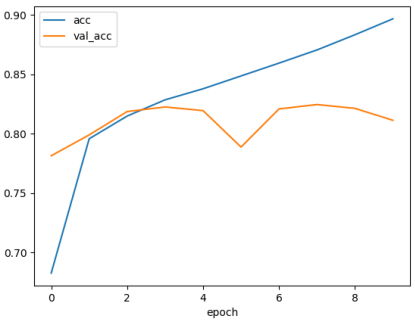
FIGURE 4.3: Summarized results of the first stage of the attention model hyperparameter grid search



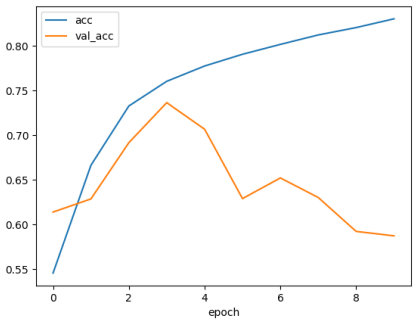
(A) Run 000



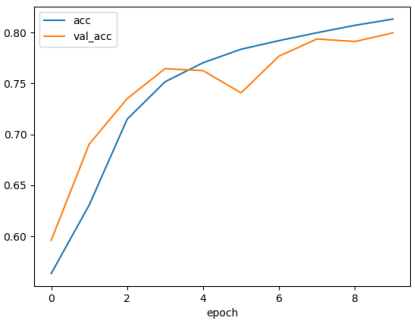
(B) Run 001



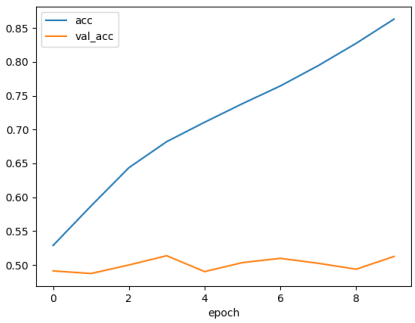
(C) Run 002



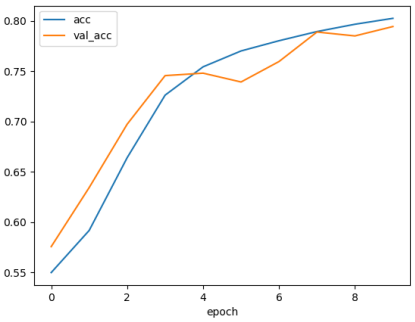
(D) Run 003



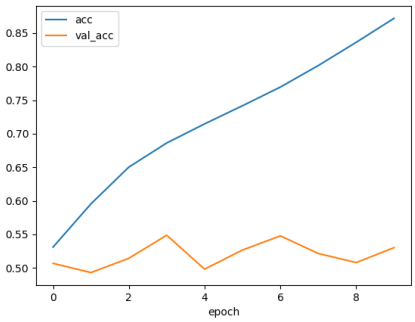
(E) Run 004



(F) Run 005



(G) Run 006



(H) Run 007

FIGURE 4.4: Accuracy training plots for each run of the first stage of the Attention model hyperparameter grid search

4.2.2 Second iteration

After the coarse grid search, we selected the best performing model and run a smaller grid search with parameters similar to those of the best run.

This time we aim to improve the results over the validation set, so we have set the stop early mechanism to track the validation accuracy.

This is the final configuration we are running:

```
data_config:
  channels:
  - image_charge
  data_type: array
  file_list_fn: data.txt
  image_mapping_config:
    hex_conversion_algorithm:
        oversampling
  img_size: [112,112]
  min_triggers_per_event: 1
  preprocessing_config:
    event_order_reverse: false
    event_order_type: size
    max_imgs_per_seq: 8
    min_imgs_per_seq: 1
    normalization: null
    resize_mode: interpolate
    sequence_padding: pre
    sequence_truncating: post
  selected_tel_types: [MSTF]
model_config:
  activation_function: relu
  cnn_layers:
  - filters: 32
    kernel_size: 3
    use_maxpool: true
  - filters: 32
    kernel_size: 3
    use_maxpool: true
  - filters: 64
    kernel_size: 3
    use_maxpool: true
  - filters: 128
    kernel_size: 3
    use_maxpool: true
  combine_mode: attention
  multi_dropout_rate:
    [0.0, 0.5, 0.8]
  fcn_layers:
  - units: 1024
  - units: 512
  input_shape: [null,112, 112, 1]
  multi_l2_regularization:
    [0.0, 1.0e-04]
  lstm_units: null
  num_classes: 2
  use_batchnorm: false
train_config:
  batch_size: 16
  class_weight: false
  decay: 0.0
  epochs: 10
  epsilon: 1.0e-08
  fit_batch_first: false
  learning_rate: 0.0001
  loss: categorical_crossentropy
  metrics: [acc, auc]
  min_delta: 0
  optimizer: adam
  patience: 3
  save_model: false
  seed: 1111
  shuffle: true
  stop_early: loss
  train_split: 0.9
  val_split: 0.1
```

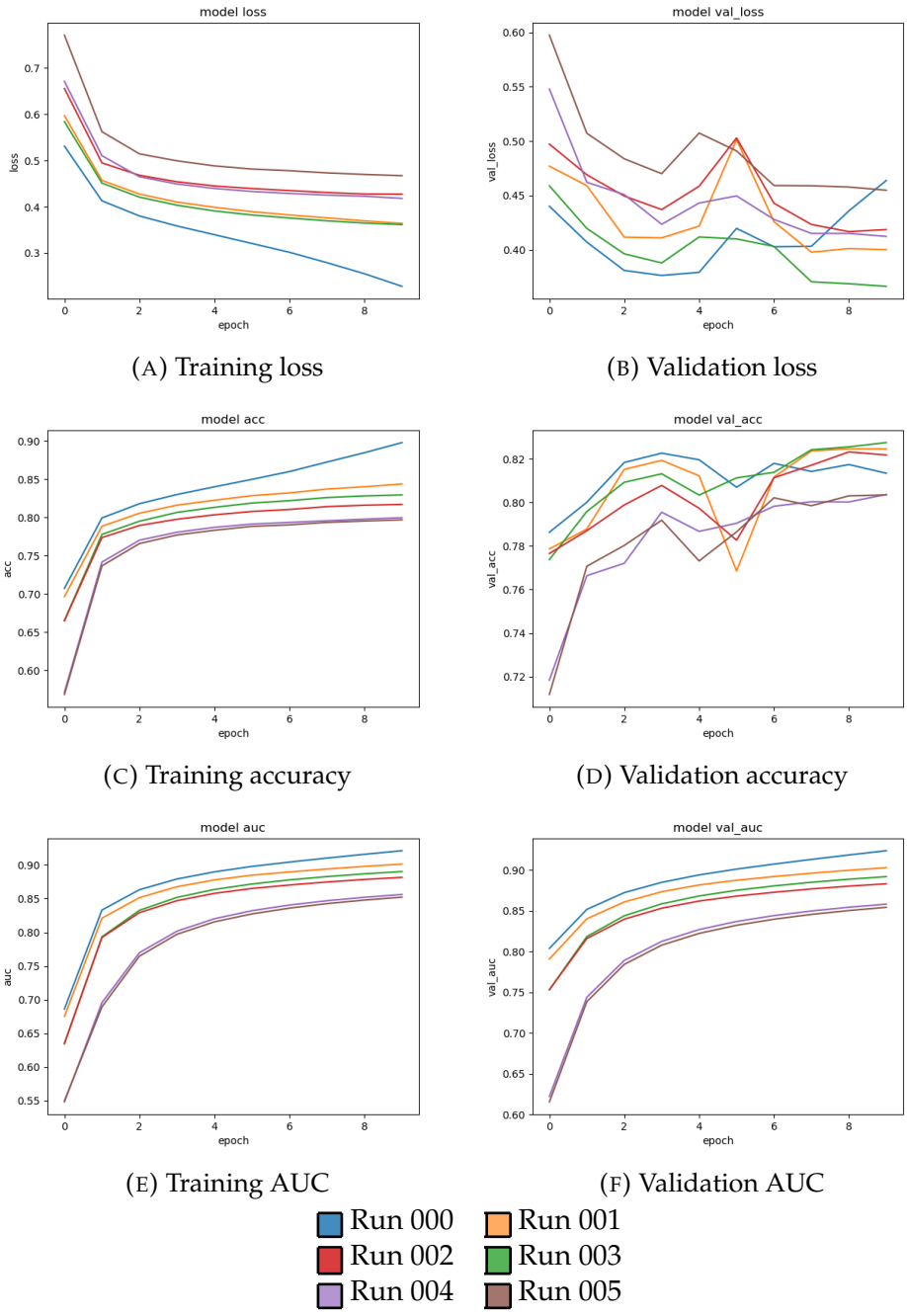
The defining characteristics of each run are summarized in table 4.2.

After the experiments finished we found the results summarized in figure 4.5.

We can appreciate better how the regularization mechanisms affected training by looking at the accuracy training plots of each run in figure 4.6.

Run Number	Dropout	L2 Weight
000	0.0	0.0
001	0.0	1e-04
002	0.5	0.0
003	0.5	1e-04
004	0.8	0.0
005	0.8	1e-04

TABLE 4.2: Distinguishing hyperparameters of each run of the second stage of the Attention grid search



(A) Training loss

(B) Validation loss

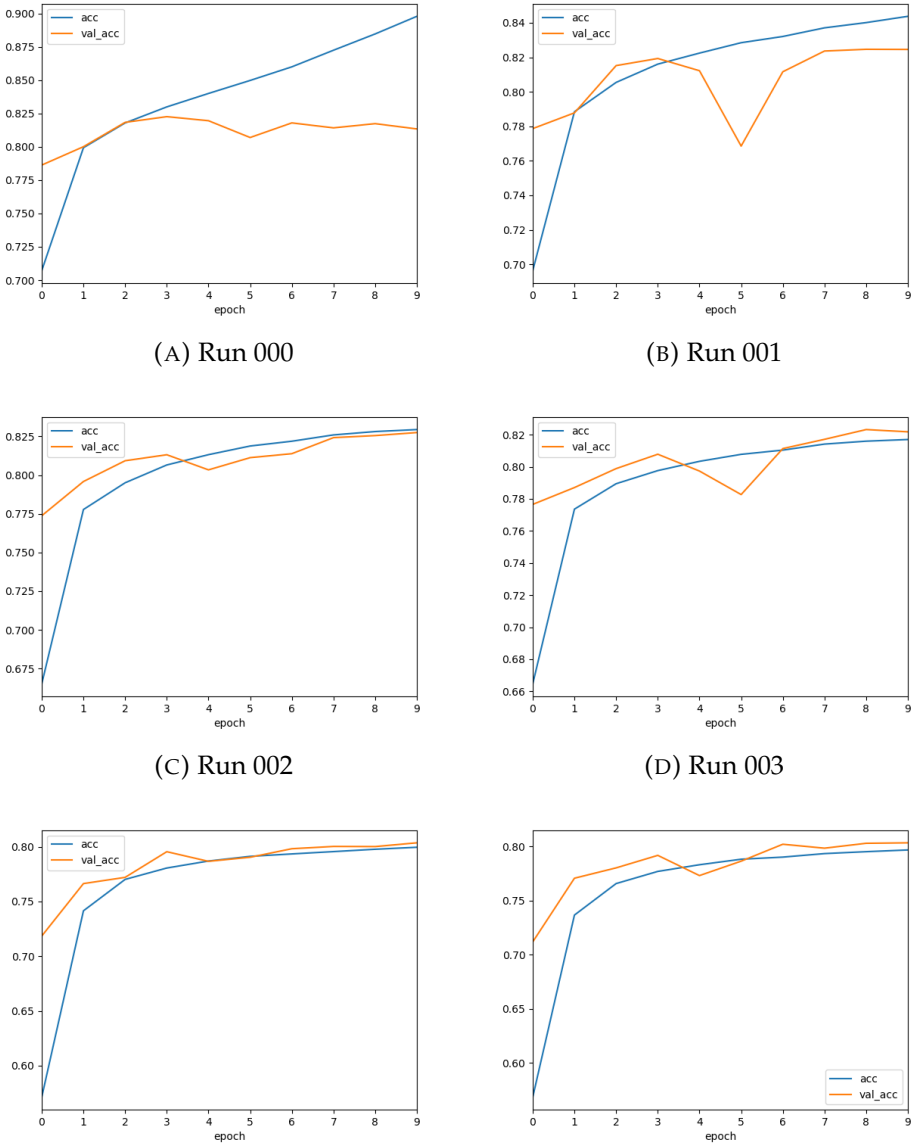
(C) Training accuracy

(D) Validation accuracy

(E) Training AUC

(F) Validation AUC

FIGURE 4.5: Summarized results of the second stage of the attention model hyperparameter grid search



(A) Run 000

(B) Run 001

(C) Run 002

(D) Run 003

(E) Run 004

(F) Run 005

FIGURE 4.6: Accuracy training plots for the second stage of the Attention model hyperparameter grid search

4.2.3 Benchmarking

Lastly we will benchmark the best performing hyperparameters we have found against all telescope types.

The configuration file used is reproduced next:

```
data_config:
  channels:
  - image_charge
  data_type: array
  file_list_fn: data.txt
  image_mapping_config:
    hex_conversion_algorithm:
        oversampling
  img_size: [112, 112]
  min_triggers_per_event: 1
  preprocessing_config:
    event_order_reverse: false
    event_order_type: size
    max_imgs_per_seq: 8
    min_imgs_per_seq: 1
    normalization: null
    resize_mode: interpolate
    sequence_padding: pre
    sequence_truncating: post
  multi_selected_tel_types:
    [[LST], [MSTF], [MSTN], [MSTS],
     [SST1], [SSTA], [SSTC]]
model_config:
  activation_function: relu
  cnn_layers:
  - filters: 32
    kernel_size: 3
    use_maxpool: true
  - filters: 32
    kernel_size: 3
    use_maxpool: true
  - filters: 64
    kernel_size: 3
    use_maxpool: true
  - filters: 128
    kernel_size: 3
    use_maxpool: true
  combine_mode: attention
  dropout_rate: 0.5
  fcn_layers:
  - units: 1024
  - units: 512
  input_shape: [null, 112, 112, 1]
  l2_regularization: 0.0
  lstm_units: null
  num_classes: 2
  use_batchnorm: false
train_config:
  batch_size: 16
  class_weight: false
  decay: 0.0
  epochs: 10
  epsilon: 1.0e-08
  fit_batch_first: false
  learning_rate: 0.0001
  loss: categorical_crossentropy
  metrics:
  - acc
  - auc
  min_delta: 0
  optimizer: adam
  patience: 3
  save_model: false
  seed: 1111
  shuffle: true
  stop_early: loss
  train_split: 0.9
  val_split: 0.1
```

The final results on validation are compiled in table 4.3.

Telescope type	Events	Positive	Negative	AUROC	Accuracy	Training time
LST	8541	3884	4656	0.8084	75.25%	0:39:52
MSTF	24982	12122	12859	0.8922	82.45%	2:37:11
MSTN	26937	13037	13899	0.9208	85.78%	3:03:10
MSTS	22306	11091	11214	0.9098	83.94%	3:06:59
SST1	19788	10337	9451	0.8809	80.87%	1:31:53
SSTA	18367	9445	8922	0.8838	82.16%	1:35:44
SSTC	19064	9789	9275	0.9093	84.53%	1:33:37

TABLE 4.3: Validation results for Attention model, v0.2.0 benchmark

4.3 Multiple telescopes

Our new framework includes a functionality not present in the v0.2.0 CTLearn framework: training with data from multiple telescopes.

We have configured a run with this feature.

```
data_config:
  channels:
  - image_charge
  data_type: array
  file_list_fn: data.txt
  image_mapping_config:
    hex_conversion_algorithm:
        oversampling
  img_size:
  - 112
  - 112
  min_triggers_per_event: 1
  preprocessing_config:
    event_order_reverse: false
    event_order_type: size
    max_imgs_per_seq: 8
    min_imgs_per_seq: 1
    normalization: null
    resize_mode: interpolate
    sequence_padding: pre
    sequence_truncating: post
  selected_tel_types:
    [LST, MSTF, MSTN, MSTS,
     SSTC, SST1, SSTA]
train_config:
  batch_size: 16
  class_weight: false
  decay: 0.0
  epochs: 10
  epsilon: 1.0e-08
  fit_batch_first: false
  learning_rate: 0.0001
  loss: categorical_crossentropy
  metrics:
  - acc
  - auc
  min_delta: 0
  optimizer: adam
  patience: 1
  save_model: false
  seed: 1111
  shuffle: true
  stop_early: loss
  train_split: 0.9
  val_split: 0.1
model_config:
  activation_function: relu
  cnn_layers:
  - filters: 32
    kernel_size: 3
    use_maxpool: true
  - filters: 32
    kernel_size: 3
    use_maxpool: true
  - filters: 64
    kernel_size: 3
    use_maxpool: true
  - filters: 128
    kernel_size: 3
    use_maxpool: true
  combine_mode: attention
  dropout_rate: 0.5
  fcn_layers:
  - units: 1024
  - units: 512
  input_shape: [null, 112, 112, 1]
  l2_regularization: 0.0
  lstm_units: null
  num_classes: 2
  use_batchnorm: false
```

In this mode, for each event we feed to our model images from all telescopes, ordered by their brightness. For events with more than 8 images we only feed to the model the 8 brightest (albeit in principle we could feed it as many as we'd like).

See the loss, accuracy and AUROC training graphs in figure 4.7.

The results of the run are summarized in table 4.4.

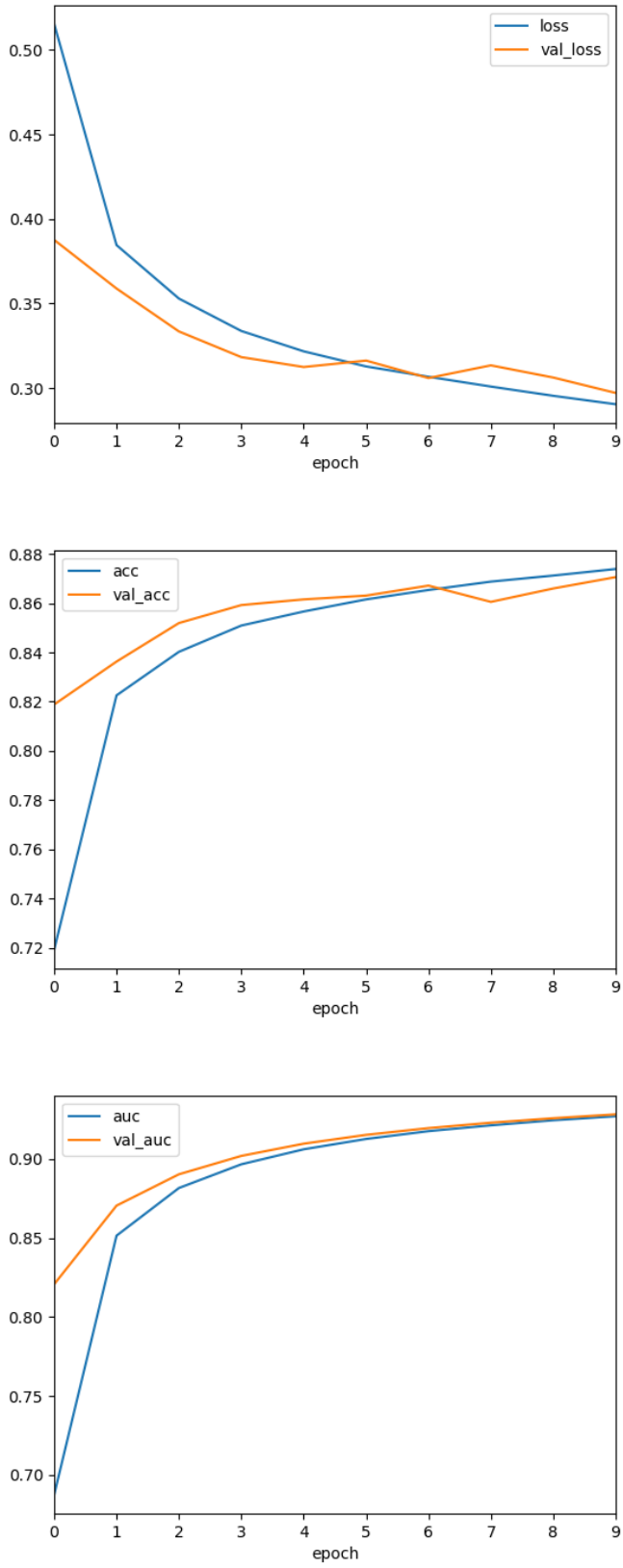


FIGURE 4.7: training plots of the attention model results trained on all telescopes

Telescope type	Events	Positive	Negative	AUROC	Accuracy	Training time
ALL	37710	18941	18769	0.9283	87.06%	15:31:58

TABLE 4.4: Validation results for attention model trained on all telescopes

Looking at the plots, the results look quite encouraging, with a high performance and no signs of overfitting.

	CNN-RNN			Attention		
Telescope type	AUROC	Accuracy	Training time	AUROC	Accuracy	Training time
LST	0.8285	73.43%	0:37:01	0.8084	75.25%	0:39:52
MSTF	0.8961	80.23%	2:06:32	0.8922	82.45%	2:37:11
MSTN	0.9169	83.10%	2:15:52	0.9208	85.78%	3:03:10
MSTS	0.9048	81.18%	2:37:34	0.9098	83.94%	3:06:59
SST1	0.8997	81.47%	3:21:48	0.8809	80.87%	1:31:53
SSTA	0.8556	75.27%	2:10:55	0.8838	82.16%	1:35:44
SSTC	0.9072	80.64%	1:51:05	0.9093	84.53%	1:33:37

TABLE 4.5: Comparison of CNN-RNN model and attention model, v0.2.0 benchmark

4.4 Attention experiments conclusion

We have trained an Attention model via iterative grid search, and we have concluded that the best results are obtained with a moderate initial learning rate (1e-4), no batchnorm, some moderate dropout (0.5) and no L2 regularization. Furthermore we have learned that fitting a single batch first does not make any significant difference.

Surprisingly the configuration that worked best with the LSTM also happened to be the best configuration for the attention model, pointing in the direction that these observations can generalize to other models.

If we compare the results of our Attention model with CTLearn's CNN-RNN model (see table 4.5) we realize that we are getting fairly similar results.

Most notable exceptions are the results over the SSTA and SSTC, where the Attention model is able to outperform significantly the CNN-RNN baseline.

On the other cases we can observe that we are getting slightly better results on accuracy but slightly worse results on AUC, which makes sense given that we used accuracy as our optimization target metric during the grid search rather than AUROC.

Finally, we notice that using data from all telescopes results in vastly better accuracy performance, from the 83.10% accuracy using MSTN data to 87.06% accuracy in the all telescopes mode, at the cost of a vastly increased training time.

This is further evidence in the direction of ensemble models using data from many telescopes being key for effective classification.

Chapter 5

Conclusion

5.1 Results

We have updated a framework for the development of IACT Deep Learning models, and developed two new models for this task, based respectively on LSTMs and neural attention.

We have carefully trained both models via backpropagation and iterated grid search, and compared the results with the existing CNN-RNN model by CTLearn in the v0.2.0 benchmark. The final results on data from the 0.2.0 CTLearn benchmark are summarized on table 5.1.

Both models have matched and in some cases like the SSTA and SSTC greatly exceeded the existing CNN-RNN benchmark. Furthermore, since the LSTM and attention model can process arbitrarily many images without the need of retraining, they can theoretically be used with sequences of data of any length to get potentially better results than with the previously existing models.

Between the two models we observe that the Attention model gets slightly better results overall with the same amount of training. As a trade-off, the Attention model takes longer to train, and its memory usage grows linearly with the sequence length, while the LSTM model uses constant memory and is thus better suited to process extremely long sequences of images.

The best results we have obtained correspond to using the attention model using data from all telescopes, and those results vastly outperform the existing benchmark established by CTLearn and the rest of our experiment, at least according to the accuracy metric. The second-best results were obtained by the LSTM model

	LSTM			Attention		
Telescope type	AUROC	Accuracy	Training time	AUROC	Accuracy	Training time
LST	0.8203	74.06%	0:41:12	0.8084	75.25%	0:39:52
MSTF	0.8837	81.62%	1:32:50	0.8922	82.45%	2:37:11
MSTN	0.9177	85.43%	2:07:06	0.9208	85.78%	3:03:10
MSTS	0.9071	83.58%	2:28:27	0.9098	83.94%	3:06:59
SST1	0.8906	80.88%	1:45:08	0.8809	80.87%	1:31:53
SSTA	0.8782	81.25%	1:11:59	0.8838	82.16%	1:35:44
SSTC	0.9157	84.59%	1:34:03	0.9093	84.53%	1:33:37
ALL	0.9298	85.77%	12:54:10	0.9283	87.06%	15:31:58

TABLE 5.1: Final results of LSTM model and attention model, v0.2.0 benchmark

using also data from all telescopes. This performance improvement suggests that ensemble models combining data from multiple telescopes matters more to higher performance than the architecture choice.

For reproducibility purposes, the code and configuration files for all our experiments can be found in https://github.com/Jsevillamol/ctlearn.

During training we have learned some further lessons about training neural networks that may generalize to other problems. In particular, we have observed that:

- overfitting first the neural network to a single batch makes no difference to the end result of training
- batchnorm makes training unstable
- excessive regularization worsens the results, but some moderate dropout helps bridge the gap between training and validation results

5.2 Outlook

Our models have been trained using grid search, which is a fairly unsophisticated training strategy. Better search schemes for hyperparameter optimization may yield better results. Some possibilities are **genetic algorithms** [Mit98] or **neural architecture search** [EMH19].

Our grid search was also quite limited in its scope, and we have not explored many relevant parameters such as the number of filters per layer, the number of layers or different hexagonal conversion algorithms.

Another aspect we have not fully explored is exactly how much results can be improved by allowing significantly larger sequences of data, as our models were artificially capped to process 8 images per event. In this context, exploring the memory requirements of the LSTM and attention model will be interesting to understand better their trade-offs.

Another route worth undertaking is developing better visualization tools to understand how the current models are being trained and make their predictions. The attention model lends itself to a visualization of how it **assigns importance** to each image when making a prediction [Vig19], and visualizing how the LSTM model trains its weights could help detect **vanishing gradients** problems [KK01].

The Deep Learning framework we developed could use better functionality to allow a better user experience for training custom models. Once a stable version of Tensorflow 2.0 is released we could update the repository to use it. Tensorflow 2.0 includes some attractive features for us, including better support for the Keras API [Ten19].

Bibliography

[AB18] Qi Feng Daniel Nieto Jaime Sevilla Ari Brill Bryam Kim. *CTLearn: Deep Learning for IACT Event Reconstruction*. 2018. URL: https://github.com/ctlearn-project/ctlearn.

[Aba+15] Martín Abadi et al. *TensorFlow: Large-Scale Machine Learning on Heterogeneous Systems*. Software available from tensorflow.org. 2015. URL: https://www.tensorflow.org/.

[Ber+13] K. Bernlöhr et al. "Monte Carlo design studies for the Cherenkov Telescope Array". In: *Astroparticle Physics* 43 (2013). Seeing the High-Energy Universe with the Cherenkov Telescope Array - The Science Explored with the CTA, pp. 171 –188. ISSN: 0927-6505. DOI: https://doi.org/10.1016/j.astropartphys.2012.10.002. URL: http://www.sciencedirect.com/science/article/pii/S0927650512001867.

[Bri18] Ari Brill. *Deep Learning for IACT Event Reconstruction*. 2018.

[Cho+14] Kyunghyun Cho et al. "Learning Phrase Representations using RNN Encoder-Decoder for Statistical Machine Translation". In: *CoRR* abs/1406.1078 (2014). arXiv: 1406.1078. URL: http://arxiv.org/abs/1406.1078.

[Cho15] François Chollet. *keras*. https://github.com/fchollet/keras. 2015.

[Cs2] *CS231n Convolutional Neural Networks for Visual Recognition*. URL: http://cs231n.github.io/.

[Cta] *Cherenkov Telescope Array - Exploring the Universe at the Highest Energies*. URL: https://www.cta-observatory.org/.

[EMH19] Thomas Elsken, Jan Hendrik Metzen, and Frank Hutter. "Neural Architecture Search: A Survey". In: *Journal of Machine Learning Research* 20 (2019), 55:1–55:21. URL: http://jmlr.org/papers/v20/18-598.html.

[HS97] Sepp Hochreiter and Jürgen Schmidhuber. "Long Short-Term Memory". In: *Neural Comput.* 9.8 (Nov. 1997), pp. 1735–1780. ISSN: 0899-7667. DOI: 10.1162/neco.1997.9.8.1735. URL: http://dx.doi.org/10.1162/neco.1997.9.8.1735.

[IL73] A.G. Ivakhnenko and V.G. Lapa. *Cybernetic Predicting Devices*. Jprs report. CCM Information Corporation, 1973. URL: https://books.google.es/books?id=FhwVNQAACAAJ.

[KB14] Diederik P. Kingma and Jimmy Ba. *Adam: A Method for Stochastic Optimization*. cite arxiv:1412.6980Comment: Published as a conference paper at the 3rd International Conference for Learning Representations, San Diego, 2015. 2014. URL: http://arxiv.org/abs/1412.6980.

[KK01] J. F. Kolen and S. C. Kremer. "Gradient Flow in Recurrent Nets: The Difficulty of Learning LongTerm Dependencies". In: (2001). DOI: 10.1109/9780470544037.ch14. URL: https://ieeexplore.ieee.org/document/5264952.

[KPM17] M. Krause, E. Pueschel, and G. Maier. "Improved γ/hadron separation for the detection of faint γ-ray sources using boosted decision trees". In:

Astroparticle Physics 89 (Mar. 2017), pp. 1–9. DOI: 10.1016/j.astropartphys.2017.01.004. arXiv: 1701.06928 [astro-ph.IM].

[KSH12] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E. Hinton. "ImageNet Classification with Deep Convolutional Neural Networks". In: NIPS'12 (2012), pp. 1097–1105. URL: http://dl.acm.org/citation.cfm?id=2999134.2999257.

[Mit98] Melanie Mitchell. *An Introduction to Genetic Algorithms*. Cambridge, MA, USA: MIT Press, 1998. ISBN: 0262631857.

[Nie+17] D. Nieto et al. "Exploring deep learning as an event classification method for the Cherenkov Telescope Array". In: (2017). eprint: arXiv:1709.05889.

[Nie18] Michael A. Nielsen. *Neural Networks and Deep Learning*. Sept. 2018. URL: http://neuralnetworksanddeeplearning.com/.

[OvE09] S. Ohm, C. van Eldik, and K. Egberts. "γ/hadron separation in very-high-energy γ-ray astronomy using a multivariate analysis method". In: *Astroparticle Physics* 31 (June 2009), pp. 383–391. arXiv: 0904.1136.

[Rad+18] Alec Radford et al. "Language Models are Unsupervised Multitask Learners". In: (2018). URL: https://d4mucfpksywv.cloudfront.net/better-language-models/language-models.pdf.

[Rad+19] Alec Radford et al. "Language Models are Unsupervised Multitask Learners". In: (2019).

[Rog16] Alex Rogozhnikov. *Gradient Boosting explained*. June 2016. URL: http://arogozhnikov.github.io/2016/06/24/gradient_boosting_explained.html.

[Sil+16] David Silver et al. "Mastering the Game of Go with Deep Neural Networks and Tree Search". In: *Nature* 529.7587 (Jan. 2016), pp. 484–489. ISSN: 0028-0836. DOI: 10.1038/nature16961.

[Sri+14] Nitish Srivastava et al. "Dropout: A Simple Way to Prevent Neural Networks from Overfitting". In: *Journal of Machine Learning Research* 15 (2014), pp. 1929–1958. URL: http://jmlr.org/papers/v15/srivastava14a.html.

[SVL14] Ilya Sutskever, Oriol Vinyals, and Quoc V. Le. "Sequence to Sequence Learning with Neural Networks". In: *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*. NIPS'14. Montreal, Canada: MIT Press, 2014, pp. 3104–3112. URL: http://dl.acm.org/citation.cfm?id=2969033.2969173.

[Ten19] Tensorflow. *What's coming in TensorFlow 2.0*. medium.com [Online; posted 14-January-2019]. Jan. 2019.

[Vas+17] Ashish Vaswani et al. *Attention Is All You Need*. 2017. eprint: arXiv:1706.03762.

[Vig19] Jesse Vig. "Visualizing Attention in Transformer-Based Language Representation Models". In: *CoRR* abs/1904.02679 (2019). arXiv: 1904.02679. URL: http://arxiv.org/abs/1904.02679.

[Vin+19] Oriol Vinyals et al. *AlphaStar: Mastering the Real-Time Strategy Game StarCraft II*. https://deepmind.com/blog/alphastar-mastering-real-time-strategy-game-starcraft-ii/. 2019.

[Yan16] Shi Yan. *Understanding LSTM and its diagrams – ML Review – Medium*. Mar. 2016. URL: https://medium.com/mlreview/understanding-lstm-and-its-diagrams-37e2f46f1714.

[Alb+08] J. Albert et al. "Implementation of the Random Forest method for the Imaging Atmospheric Cherenkov Telescope MAGIC". In: *Nuclear Instruments and Methods in Physics Research A* 588 (Apr. 2008), pp. 424–432. DOI: 10.1016/j.nima.2007.11.068. arXiv: 0709.3719.

[Bec+11] Y. Becherini et al. "A new analysis strategy for detection of faint γ-ray sources with Imaging Atmospheric Cherenkov Telescopes". In: *Astroparticle Physics* 34 (July 2011), pp. 858–870. DOI: 10.1016/j.astropartphys.2011.03.005. arXiv: 1104.5359 [astro-ph.HE].

[Shi+18] I. Shilon et al. "Application of Deep Learning methods to analysis of Imaging Atmospheric Cherenkov Telescopes data". In: *ArXiv e-prints* (Mar. 2018). arXiv: 1803.10698 [astro-ph.IM].